NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DEVELOPMENT OF INFORMATION ASSURANCE PROTOCOL FOR LOW BANDWIDTH NANOSATELLITE COMMUNICATIONS

by

Cervando A. Banuelos II

September 2017

Thesis Advisor: Marcus Stefanou
Co-Advisor: Jim Horning

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2017	Master's thesis

4. TITLE AND SUBTITLE
DEVELOPMENT OF INFORMATION ASSURANCE PROTOCOL FOR LOW BANDWIDTH NANOSATELLITE COMMUNICATIONS

5. FUNDING NUMBERS

6. AUTHOR(S) Cervando A. Banuelos II

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

Nanosatellites provide a light, efficient, and cost-effective way for research institutions to carry out experiments in low Earth orbit. These satellites frequently use the ultra-high and very high frequency bands to transfer their data to the ground stations, and oftentimes will use internet protocol and Transmission Control Protocol as a standard for communication to ensure the arrival and integrity of the data transmitted. Due to bandwidth limitations and signal noise, these connection-based protocols end up accruing a large data bandwidth cost in headers and retransmissions. Furthermore, due to connection unreliability, encryption and integrity checks present a challenge.

The aim of this thesis is to develop a software-based low-bandwidth reliable network protocol that can support a cryptographic system for encrypted communications using commercial off-the-shelf components. This protocol reduces the data overhead, retains the retransmission functionality and integrates support for a cryptographic system. This thesis develops the encryption mechanism, assesses its resilience to error propagation, and develops the protocol to work over a simulated network. The result of the study is a proof of concept that the protocol design is feasible, applicable, and could be used as a communication standard in future projects.

14. SUBJECT TERMS	15. NUMBER OF PAGES
commercial off-the-shelf technology, nanosatellites, CubeSat, encrypted communications	161
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

DEVELOPMENT OF INFORMATION ASSURANCE PROTOCOL FOR LOW BANDWIDTH NANOSATELLITE COMMUNICATIONS

Cervando A. Banuelos II
Civilian, National Science Foundation CyberCorps
B.S., Texas A&M University, 2013

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2017

Approved by: Marcus Stefanou, Ph.D.
Thesis Advisor

Jim Horning
Co-Advisor

Peter Denning, Ph.D.
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Nanosatellites provide a light, efficient, and cost-effective way for research institutions to carry out experiments in low Earth orbit. These satellites frequently use the ultra-high and very high frequency bands to transfer their data to the ground stations, and oftentimes will use internet protocol and Transmission Control Protocol as a standard for communication to ensure the arrival and integrity of the data transmitted. Due to bandwidth limitations and signal noise, these connection-based protocols end up accruing a large data bandwidth cost in headers and retransmissions. Furthermore, due to connection unreliability, encryption and integrity checks present a challenge.

The aim of this thesis is to develop a software-based low-bandwidth reliable network protocol that can support a cryptographic system for encrypted communications using commercial off-the-shelf components. This protocol reduces the data overhead, retains the retransmission functionality and integrates support for a cryptographic system. This thesis develops the encryption mechanism, assesses its resilience to error propagation, and develops the protocol to work over a simulated network. The result of the study is a proof of concept that the protocol design is feasible, applicable, and could be used as a communication standard in future projects.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

3DES	Data Encryption Standard
AES	128 bit Advanced Encryption Standard
BER	bit error rate
COTS	commercial off-the-shelf
CRC32/CRC	32-bit cyclic redundancy check
CSP	CubeSat Space Protocol
HMAC	keyed-hash message authentication code
IP	Internet Protocol
IPv4	Internet Protocol version 4
ISO	International Organization for Standardization
LEO	low Earth orbit
MEROPE	Montana EaRth Orbiting Pico Explorer
NERDP	Nanosatellite Encrypted Reliable Datagram Protocol
OSI	Open Systems Interconnection
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UHF	ultra-high frequency
VHF	very-high frequency
XOR	exclusive or logical function

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First and foremost, a big thank you to Dr. Marcus Stefanou and Jim Horning for believing in me and my project, and supporting me in every possible way. I came to you with a vague idea and a limited amount of time, and with your hard work, motivation, and direction, you helped me reach new formidable heights. Thank you also to the Computer Science Department and the Space Systems Academic Group at the Naval Postgraduate School for providing me with the resources and education to make this possible.

To my parents and sister whose constant encouragement and care packages fueled the hard work and writing that went into this thesis, I want to say thank you for believing in me.

Thank you to Annathea Cook for the love and support, and for the angry wake-up phone calls to make sure I made it to class on time. To DeeDee and Liz, thanks for your support, encouragement, and for all the times I needed someone to talk to. Thanks Dan Gifford for helping come up with the topic of my thesis. And a big thank you to Jay Ellis for feeding me when I was hungry and unable to afford food.

Special thanks to Devon Ellis for listening to my crazy ideas, for helping me become the computer scientist I am today, and for helping me find bugs I wouldn't have otherwise been able to find in my code. Thanks also to Alexis Rogers for listening to me and for helping me find bugs in my code.

Finally, a thank you to StackOverflow. Without you, I would not have been able to survive the last two years or write this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. RESEARCH DOMAIN

Nanosatellites, small satellites, and CubeSats are small low Earth orbit (LEO) devices used to conduct space-based research in a cost effective manner. Nanosatellites typically have a mass of less than ten kilograms, have a life time of a few weeks or months on orbit, and are often constructed using commercial off-the-shelf (COTS) components. COTS components are typically inexpensive, readily available, and can be easily repurposed for space missions. The use of these components helps keep the mission costs low and allows for a larger number of research institutions to carry out experiments and demonstrations in LEO. CubeSats are satellites composed of a volume of 10x10x10 centimeter cubic units, while small satellites are all satellites with a mass less than 1000kgs [1]. Throughout this investigation, the term “nanosatellite” will largely refer to any satellite with limitations in its data transfer rate, and a mass less than ten kilograms, and a volume similar to the CubeSats. Ultimately the scope of this research is not the satellite physical configuration, but rather the power and bandwidth limitations.

Currently, nanosatellites and their COTS components rely heavily on pre-existing and well established communication protocols. These protocols are also used in ground based internet communications and build on the Internet Protocol (IP) stack. Specifically, researchers use two of the most common protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). These protocols operate at a network level on all computers on the ground, and provide a communications framework to automate the transmission and receipt of data.

TCP is a connection-based protocol, meaning that it relies heavily on a persistent connection even if the connection is noisy or prone to errors in the data. TCP provides key services that are fundamental to the reliable transmission of data such as retransmission of lost or deformed packets, acknowledgement of data received, integrity checks, and the ability to assemble the packets of data in the correct order. To achieve

this, each TCP packet contains between 20 and 60 bytes of data as a header containing the relevant information needed by the receiver to carry out these functions.

UDP is a lighter protocol that does not rely on a persistent connection. UDP is a unidirectional, meaning it requires no acknowledgement of receipt, packet sent by a transmitter to a receiver without any information for retransmission, or correct packet ordering. If an object is fragmented into discrete packets and transmitted with UDP, unlike TCP, these packets may or may not arrive, and they may or may not arrive in the right order without any mechanism to verify their order, without a mechanism to acknowledge their successful arrival to the recipient, and no way for the recipient to request the retransmission of a specific packet. UDP does provide a checksum for integrity validation of the packet, but not much more data is transmitted in its 8-byte header. This headers matter as they encapsulate all packets of data sent by the nanosatellite. Whether that data is telemetry, commands or payload data, it is all streamed as packets preceded by a packet header.

These data packets are frequently transmitted to and from nanosatellites over ultra-high and very-high frequency (UHF and VHF) bands. These radio frequencies allow researchers and the operators of the nanosatellites to communicate with the devices in orbit at a low financial cost, leveraging COTS transmitting and receiving equipment. By using these bands, nanosatellite operators can also reduce the power consumption and internal size, weight, and power requirements of nanosatellite communications components. Unlike most internet data packets sent using UDP or TCP, these packets are not encrypted due to the limitations on the nanosatellites. This lack of encryption is a security vulnerability that allows data to remain confidential.

Nanosatellites provide an accessible opportunity for more institutions to carry out space-based research. The devices have lower expenses than other space missions, are small, and the components are readily available. Since the launch of the first nanosatellites in the early 2000s, cost-benefit equation has been driven by the low cost and profile of the devices. Furthermore, the ability to transmit and receive the data from the devices is beneficial to research institutions who would otherwise have no way to extend their research projects to space exploration. To this end, it is important that a

standard in data communications for nanosatellites be established to broaden the scope of the research capabilities of the nanosatellites. The ideal standard should take into account the technical limitations of the nanosatellites, be flexible in its implementation to accommodate various nanosatellite designs, be a software based solution, and provide an efficient mechanism for communication that improves upon existing communications protocols.

B. MOTIVATION

The popularity of nanosatellites is due largely to their relative simplicity and affordability. Unfortunately, these benefits come at a cost. These costs translate to low signal-to-noise ratio, low-bandwidth, high packet drop rates, and low overall mission data transfers. These limit the range and length of experiments accessible and available, and limit the usage of well-established IP communication schemes and encryption methods

To make nanosatellites more accessible to multiple research institutions, and to simplify the communication schemes, researchers have designed nanosatellites to communicate over amateur radio bands in UHF and VHF using a variety of radio protocols. As mentioned above, the use of these protocols and these bands means that there is a relatively low data transmission rate accessible for space to ground communications. Regardless of whether these satellites use transceivers custom-built for the specific mission, or prefabricated COTS transceivers, if a common communication protocol is the AX.25 protocol.

Establishing the rate of data transfer in the UHF an VHF band is important since it is the limiting factor for all of the protocols utilized. Surveys done by two teams, Bryan Klofas et al. in 2008, and Paul Muri and Janis McNair in 2012, show that nanosatellites, specifically CubeSats operating in the UHF band, typically have a baud rate ranging from 1200–9600 symbols per second [2], [3]. Klofas' survey, dated in 2008, shows a comparison summary of the various communication transceivers, frequencies, in addition to the baud rate of 18 different satellites operating in UHF and VHF bands. This survey also specifies the data link layer protocol, Open Systems Interconnection (OSI) Layer 2 as defined by the International Organization for Standardization (ISO), used by these

satellites. More specifically, the survey shows that out of the 18 satellites included in the paper, 14 devices utilize the AX.25 protocol for amateur packet radio [2]. Muri and McNair's survey, shows a database of 30 satellites launched in the 2009–2011 timeframe; of these devices 16 utilized the AX.25 protocol [3].

The AX.25 packet radio protocol ensures the delivery of packet data encapsulated in frames and managed by the hardware in the transceiver. This protocol provides a standard for the intercommunication between various ground stations and satellites in either half or full-duplex schemes. Unfortunately, this protocol does not intrinsically provide any support for the implementation of the IP protocols such as TCP or UDP, as those operate on the OSI Layer 3, the Network Layer [4]. These protocols provide packet management at the network layer and focus on the reassembly of data before handing it off to the application layer. AX.25 is the de facto standard for communication due to its long standing history as a packet radio transfer protocol. AX.25 operates on amateur radio bands, which are ideal if communication is being done in UHF and VHF bands [4]. AX.25 operates in the data link layer and only offers functionality to deliver the packets from one end of the data stream to another. The lack of network packet management functionality provided by TCP or UDP in the AX.25 protocol means that these protocols typically have to be added on top of the existing OSI Layer 2 much like those same IP protocols have to be used in addition to the Ethernet frames in standard internet communications. TCP and UDP are not good replacements for AX.25 because while they can reassemble the data packets, they lack the mechanisms for transferring the data from one radio to another.

TCP and UDP have their drawbacks in design and applicability. TCP is heavily connection based protocol that requires a persistent, connection, ideally running in full duplex mode. This allows the transmitter to receive acknowledgements while it transmits data packets. Unfortunately, due to the limitations of the AX.25 protocol in the amount of possible data transmitted per frame, the relatively higher noise rate of the UHF and VHF bands, and the size of the TCP header, TCP become unwieldly for nanosatellites with lower baud. At 9600 baud, a nanosatellite can transmit 9600 symbols per second, and at half duplex this could present a large data cost to an already limited bandwidth.

An OSI Layer 4, Transport Layer, solution has been proposed by members of Aalborg University in Denmark called the CubeSat Space Protocol (CSP) [5]. This protocol was developed in C and modeled after the IP TCP standard and includes a header that is only 4 bytes long and supports eXtended Tiny Encryption Algorithm (XTEA) encryption and is designed to successfully integrate with several physical layer technologies. While this protocol does provide some additional functionality at a lower cost, it is limited to the specific physical layer drivers and is more centered towards network operations. This is reflected by looking at the packet structure and noticing that it uses 22 bits out of the available 32 just to establish a source, destination, and their corresponding ports [5]. Since most of the source and destination addressing can be done at the OSI Layer 2 for most radios, it is inefficient to use that much of the packet header in a redundant manner. Furthermore, CSP reserves several ports for buffer status, pings, and other network functions that may not be a priority for researchers or can again be derived from the radio protocol used. The use of XTEA does not allow partial decryption, as described above, and limits data validation to only after the entirety of the object has been downloaded.

From a security perspective, nanosatellite communication schemes lack a cryptographic method that ensures the confidentiality of the data transmitted. This presents a security vulnerability as it compromises the integrity of the data and allows unauthorized parties to view and record the data. In internet communications, this vulnerability is mitigated with encryption mechanisms that ensure the data is maintained private and only users with the appropriate keys are able to decrypt the data. While there are some solutions that provide encryption of data, such as CubeSec and GndSec solutions devised by Challa et al. in [6], these solutions are hardware based. Hardware based implementations are those that require external hardware other than the main computational processor to carry out the encryption. These implementations reduce the load on the main processor by adding smaller microcontrollers only carry out the encryption. Limiting communications to specific hardware configurations places a constraint in the design and flexibility of nanosatellites. While hardware implementation of encryption may be faster for certain encryption methods as stated in [6], a low-impact-

software encryption mechanism would be desirable as it would be independent of specific hardware constraints. An encryption mechanism called a one-time pad (OTP) is an encryption method that relies on symmetric pre-shared keys and each symbol is encrypted independently. This study hypothesizes that such an encryption scheme would provide the desirable characteristics to integrate into nanosatellite communications. This investigation will compare it to encryption methods such as the Advanced Encryption Standard (AES) in certain configurations means that if a large file is encrypted and transmitted, the receiver would have to wait to receive the whole object before decryption, which may not be in the best interests of the mission.

C. RESEARCH QUESTIONS

It is clear that there is room for improvement in the efficiency and security of nanosatellite communications. To that end, this research investigates the following questions:

1. What are the processing, data overhead, and encryption costs of current nanosatellite communication protocols? These are an aggregate of computational time, bytes of unnecessary data in headers, and complexity of encryption mechanism.

2. Is a one-time pad approach for encryption in nanosatellite communications viable, and how does this approach compare to CSP and XTEA in terms of processing and storage costs?

3. Is there a protocol scheme to reduce the amount of data overhead and result in faster transfer times and/or a reduced number of packet exchanges than TCP?

D. SCOPE

The scope of this thesis is to investigate the communication needs of the small satellite and nanosatellite community operating in the UHF and VHF bands, focusing on bandwidth limitations and developing a versatile lightweight software solution that can meet those needs and increase the productivity of the satellite. This thesis offers a new

data communications protocol called the Nanosatellite Encrypted Reliable Datagram Protocol (NERDP). This thesis will also investigate the addition of confidentiality to the data payloads using a pre-loaded one-time pad (OTP) increasing the cybersecurity strength of the communications scheme. Development will target a software solution that can be run on COTS components, measure the performance of the OTP encryption, add integrity checks for the data transmitted, and add reliability to the data transmissions while keeping hardware limitations in mind.

E. APPROACH

The process used for this investigation baselines the current performance of the transfer of nanosatellite data communications used by the Naval Postgraduate School Space Systems Academic Group, and surveys the protocols used and the challenges encountered. This study focuses primarily on the application of TCP and UDP as the main protocols for data transfer, as none of the NPS satellites currently support encryption. The NERDP prototype demonstrates TCP-like functionality in data packet reliability and retransmission at a lower cost in data and better performance in UHF and VHF. This prototype is developed as a proof of concept in a virtual network with limited applications, using a modular approach and supports the addition of increased functionality depending on mission requirements. NERDP is designed to operate strictly in OSI Layer 3 and higher, leaving the Data Link Layer to the hardware specifications and the AX.25 protocol. The Data Link Layer encapsulates the data and transports it from one point to another but does not interact with the data itself. For the information assurance component of the prototype, and independent module using OTP encryption is developed and its performance is characterized. This is done independently of the NERDP the protocol can support OTP and other types of encryption, but does not necessarily require OTP.

F. THESIS STRUCTURE

The remainder of this thesis is structured as follows:

Chapter II continues the discussion of bandwidth utilization in UHF and VHF bands, and includes a brief survey of current communication schemes and notable

nanosatellites and CubeSats relevant to this thesis. It also discusses the need for cybersecurity in nanosatellites and outlines the current state, and discusses the different methods of encryption with a particular focus on OTP.

Chapter III discusses the methodology for development, goals, and robustness of the OTP encryption algorithm designed for this thesis.

Chapter IV discusses the methodology of the development of the NERDP, the structure, reliability mechanisms, and the data overhead reduction of the Network Layer software based protocol proposed in this thesis, NERDP, and includes a comparison to other IP protocols.

Chapter V summarizes the results of the encryption scheme and NERDP as functions of overall system performance. This will evaluate the systems costs and their feasibility along with any potential cybersecurity vulnerabilities.

Chapter VI will provide conclusions about the applicability of the prototype and proposed encryption scheme, and outline the future work and next steps.

II. BACKGROUND

A. INTRODUCTION

Bandwidth limitations in the UHF and VHF bands of nanosatellite communication schemes produce a restrictive environment for the transfer of data from the spacecraft to the ground stations. The root of the issues is discussed, and a notable CubeSat is explored. These surveys provide further context of the problem space and the limitations currently encountered by nanosatellite developers. The text also provides a brief overview of cybersecurity and information assurance in nanosatellites, and a discussion on encryption with a focus the use of on one-time pads.

B. PROBLEM SPACE: LOW BANDWIDTH IN UHF AND VHF BANDS

As described in [2] and [3], most nanosatellites communicate in the UHF range and have a baud rate typically of 1200 to 9600 symbols per second. Several factors limiting this baud rate include, but are not limited to the hardware used, the power available to the communications array, antenna type, time window for communication, and angle on the horizon. Variations in all of these factors can create not only fluctuations in the baud rate but also in the quality of the signal. Lower signal quality introduces random noise and errors, typically in the form of flipped bits in the data payload, and can compromise the integrity of the overall object being transmitted. This loss of packets due to signal noise, measured as bit error rate, is part of the reason some nanosatellites use protocols like TCP or CSP as they allow for the retransmission of lost packets and packets deemed too compromised.

1. Common Nanosatellite Frequency and Bit Rate Ranges

In order to profile the communication systems of nanosatellites, a survey of their performance is presented. First the common frequency at which the se devices operate on the radio frequency spectrum must be established. The UHF an VHF bands are defined by radar-frequency letter band nomenclature, and also by the International Telecommunications Union (ITU). These nomenclatures, while similar, can lead to some

confusion. Radar nomenclature identifies the VHF band as a frequency range of 30–300 MHz, the UHF band as 200–1000MHz, the L-band as 1 -2 GHz, and the S band as 2–4 GHz. The ITU nomenclature, while maintaining the same definition of the VHF band range, groups any frequency between 300 MHz-3 GHz as UHF [7]. The surveys by Klofas et al. and Muri and McNair show that most CubeSats and nanosatellites transmit at the 435 MHz frequency [2], [3]. In the Klofas survey, of the 18 satellites examined, all but three devices operate on the range between 400.375–437.880 MHz with the outliers operating at 902–928 MHz and 2.4 GHz [2]. Muri and McNair, also showed similar results, with only ten out of the 30 satellites recorded not operating in the ~437 MHz frequency [3]. Researching this distribution further reveals that in an update to the Klofas' survey to include CubeSats launched between 2003–2014, 112 out of 172 total transmitters recorded operated in the 437 MHz amateur radio frequency range, with an additional 40 devices still operating below 1000 MHz [8].

Having established the frequency at which these devices typically operate, the data transfer rate must also be surveyed. The data transfer rate is measured in the number of bits transmitted per second (bps) or baud rate (symbols per second), and is used to determine the rate at which data can be transmitted. On ground based systems, such as the internet, speed is typically measured in the megabit range (millions of bits per second) but due to the low power and limited hardware, typical nanosatellite data transfer rate ranges typically fall into the kilobits per second range. The Klofas, and Muri and McNair surveys expose the data rates of several satellites. More specifically, out of 144 transmitters recorded by Klofas, including the other surveys, 121 transmitters operated at 9600 baud or less, with the second most common rate being 1200 bps [2], [3], [8]. These low bit rates are why these devices are labeled as low-bandwidth for the sake of this problem space and part of the reason why reducing data overhead is so important and significant.

After establishing the prevalence of the 437 MHz frequency and a typical baud rate of 9600 or less in both early and more current nanosatellites, research and development of communication protocols should strive to operate at these target specifications. These specifications seem to provide the most cost effective hardware and

communication packages for nanosatellites, as reflected by their popularity, but simultaneously also limit the usefulness of these devices. If experiments collect too much data, then it may be infeasible for the data to be transmitted to the ground recipient. The problem is exacerbated when a large portion of this limited bandwidth is needed to retransmit a large number of packets due to poor connection, where each of these packets has a large header.

2. Bit Error Rate and Packet Loss

Data rates in satellites are dictated by the communications system power, signal quality, distance between receiver and transmitter, atmospheric conditions, and other factors. These factors impact the already limited bandwidth of the COTS components in nanosatellites and introduce errors in the bits transmitted. These errors can be resolved through error correcting schemes, and through data retransmission. These unavoidable occasional retransmissions are why protocols like TCP are preferred over protocols like UDP.

Error rates in data transmissions are characterized with a measure known as bit error rate (BER). BER is defined as the ratio of incorrect bits received divided by the total number of bits transmitted. This ratio is useful in evaluating the performance of the communication system and estimating the need for retransmission and error correction. To put BER into perspective, in a 2012 report, authors Selva and Krejci utilize an estimated BER for calculations of approximately 10^{-5} [9]. This gives a base to determine the frequency of errors that occur in a given data set.

BER impacts the integrity of specific bits that are transmitted, which compromise packets. Due to the low power of the transmissions, it is also possible for packets to never reach the ground station. These total packet losses result in missing data and, in the case of TCP, result in the ground station requesting multiple retransmissions of packets. This constant change of state of the radio from receiving to transmitting accrues a time loss if the signal quality is poor enough to require multiple retransmissions. Furthermore, nanosatellites have a limited window of approximately 45 minutes of contact with the ground station per day. If changing states of the radio takes one second to transition

between states, then the two seconds it takes to request a retransmission is 0.074% of the total time available per day. If an object requires multiple retransmissions to ensure integrity, then this accrued time from state switching is detrimental to the performance of the communication system. As described above, BER is unavoidable and by consequence so are retransmissions. Therefore, to ensure optimal data transfers, a protocol that improves on the TCP model and reduces the number of state changes would provide a better solution.

C. CURRENT NANOSATELLITE COMMUNICATION STANDARDS

Current nanosatellites typically use the AX.25 packet radio protocol, and sometimes encapsulate a Layer 4 protocol such as CSP, TCP, or UDP. Each of these Layer 4 protocols has advantages, disadvantages, and applicability, but all have a data overheads required in transmission. This overhead reduces the amount of data that can be transmitted by the satellite, and adds functionality not always needed from the nanosatellite. Additionally, some of these Layer 4 protocols cause connectivity problems if the connection is unstable or unreliable and compound the problem of reliability and retransmission, further increasing the accrued data overhead.

1. Data Overhead

Due to the various designs and OSI Layer 2 implementations, such as AX.25, the calculations for optimizing data overhead focus on Layers 3 and 4. These layers, the networking and transport layers, provide the infrastructure for transferring data packets, and for dictating their behavior. In typical internet applications, Layer 3 is responsible for routing and packet forwarding structures like IPv4, while Layer 4 provides the architecture for the connection behavior in protocols such as TCP and UDP. Anything higher than Layer 4 in nanosatellites, can be considered payload data, though it should be noted that the header of Layers 3–4 is often included as part of the payload along with Layers 5 and above when viewed in reference to the Layer 2 protocol.

Since nanosatellites use AX.25 for the delivery of packets, and are largely point-to-point communication schemes, a networking layer that includes routing information is unnecessary as this layer can be used to route packets to various IP addresses in the same

network, and even make the transition through different routers. Point-to-point communication through packet radio carried out through AX.25 does not require routing or communication with multiple nodes, therefore the implementation of a header, such as an IPv4 is not necessary and abandoning it can reduce the overhead by 20–60 bytes [10].

Abandoning the need for a Layer 3 protocol introduces some challenges for IP based transport layer protocols. TCP is reliant on a persistent IPv4 based connection, and its data header includes information on the source and destination IP addresses and ports. This information supporting the range of functionality of TCP results in a header of 20–60 bytes [11]. Using the above information, a transport layer protocol that is independent of the network layer can reduce the data overhead of each packet transmitted by 40–120 bytes. In a given packet of 77 bytes, this overhead accounts for nearly 52% of the data packet. In relation to packet loss and retransmission, the costs of IP/TCP overhead accrue quickly. A time diagram of TCP transmission with packet loss, either from integrity failure or packet drop, (Figure 1) demonstrates the overly verbose nature of TCP that leads to a large amount of packet transmissions.

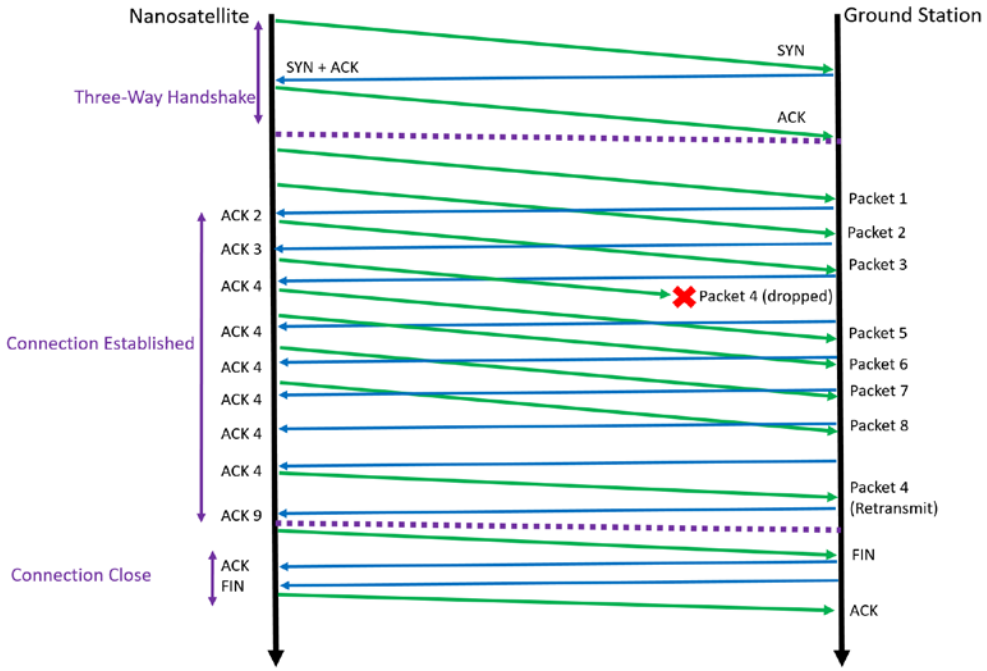


Figure 1. TCP time diagram for transmission of 8 packets with retransmission of packet 4

Each packet transmitted under TCP will have an IP header and a TCP header. Assuming both headers are their smallest possible values, the total headers for Layers 3–4 in this scheme is 40 bytes, or 320 bits, per packet. In 8 packets this data accounts for 2560 bits. At 9600 bits per second the data overhead accounts for 26.7% of the data transmitted per second, assuming the baud rate is negligibly affected by the Layer 2 AX.25 protocol.

UDP, the other popular IP protocol in nanosatellites, is a connectionless protocol that still relies on the IP infrastructure of Layer 3. This Layer 4 protocol uses one-way datagrams, a basic transfer of data unit consisting of a packet header and payload, to transmit data between two nodes. These datagrams provide a header per packet that includes source and destination ports, much like TCP, and also provides and integrity check for the data transmitted. The drawbacks of this protocol include the lack of functionality for retransmission and correct packet assembly order. TCP utilizes sequence numbers in the headers to assemble the packets in the correct order and detect if a packet is dropped. UDP's lack of sequence check creates a challenge for data retransmission and

object reconstruction. TCP also provides functionality to ensure the delivery of the packets through the form of acknowledgements per packet, while UDP has no such mechanisms. The advantage of UDP is its simplicity and significantly smaller header than TCP. Assuming 20 bytes are still used for the IP header, UDP only requires an additional 8 bytes as a header as opposed to the 20 required by TCP [12].

It should also be noted that the standards for both UDP and TCP outline 2 bytes for each of the destination and source ports in the protocol. These two bytes, or 16 bits, are unsigned integers and result in 2^{16}, or 65536, possible ports for data receipt and transmission [11], [12]. Such a large number of ports is useful in internet and network communications, but may be excessive for use in nanosatellites. A protocol with a reduced number of ports would reduce the overhead in headers at little to no cost in functionality.

CSP is a protocol designed specifically to be used with nanosatellites and CubeSats. This protocol provides support for integrity checks through a 32 bit cyclic redundancy check (CRC32 or CRC) and keyed-hash message authentication codes (HMAC), flags to signal if packets are encrypted, and 12 bits for destination and source port assignments (2^6 = 64 possible ports) [5]. This functionality is all outlined in the protocol header which is only 4 bytes, 32 bits, long. CSP provides retransmission functionality and encryption support, and can be used independently from an IP layer. This reduces a header of 40 bytes of IP/TCP by 90% to only 4 bytes.

Looking closer at the mechanisms of CSP, it becomes evident that the 4-byte header is deceptive. The header itself only contains a single bit flag denoting if the packet is encrypted, if a CRC is included in the payload, or if the packets have an HMAC, without containing any of these checks within the header itself [5]. If a packet is designated with a CRC32 then the payload data will include 4 additional bytes of information doubling this “non-payload” overhead; similarly if a packet is flagged to contain an HMAC, this will add two bytes of data to the overhead potentially increasing the header from four bytes to ten [5]. Additionally, the documentation of CSP is unclear how much overhead the retransmission infrastructure would add to the total overhead. The reason for this is unclear, and may be lack of support and use of CSP.

Data overhead is important in situations where the baud rate is limited to a noisy and error prone 9600 symbols per second. While land based communications can reliably use TCP and UDP for IP based communication, the overhead accrued with them is too high for a limited connection found in nanosatellites. These protocols also provide unused functionality in point-to-point connections that results in additional space that could be better utilized by the protocol. Other protocols like CSP promise small headers and increased functionality, but upon closer inspection fail to disclose the structure and variable “non-payload” data accrued in their functions. This data overhead in turn, while still lighter than IP-based protocols, still leaves room for improvement in reducing the overhead.

2. Connection Issues

While all protocols discussed suffer from connection issues such as error rates and packet loss, the delay in packet transmission and acknowledgement of receipt in TCP creates a specific problem that is exacerbated by the potential delays in transmission of packets. Due to the distance between nanosatellites and their ground stations, the fleeting window for transmission, and the delays in change of state in the radio hardware, there is a possibility that the TCP connection times out from inactivity or failure to receive the proper acknowledgement. Figure 1 demonstrates the state dependency of TCP, which can have a negative impact on the performance of the system.

While TCP timeouts can be set by the user to extend or shorten the time sent packets can remain unacknowledged until the connection times out and terminates [13], these values are user defined and can vary from application to application. Nanosatellite designers could decide to implement the IP/TCP model on the AX.25, as described above, with a long TCP timeout wait to ensure the connections are not dropped. This creates the problem of resource allocation and the state dependency of TCP. If a connection is kept alive for too long, there is the possibility of resource exhaustion since all of the resources will have to be allocated and maintained. The constant change of TCP between packets and acknowledgements can also create a resource allocation problem where power consumption and time are excessively consumed. Conversely, if the TCP

timeout is set too short, there is the possibility of connection timeout any time the nanosatellite loses connection with the ground station or the connection is poor. If a connection times out, the connection must be reestablished through a three-way handshake, and the file download must be restarted. These increase the data overhead, and detract from the useful windows of the nanosatellite.

UDP does not suffer from this problem of timeout and reliance on persistent network connectivity nor does it rely on the state of the transmitter and receiver, but also does not have any higher functionality. The documentation is unclear on whether or not CSP employs a connection timeout, nor does it divulge how communications are initialized in comparison to the TCP three-way handshake. Regardless, a protocol designed to take the state of the transmitter and receiver into account, carefully weigh the limitations and benefits of a connection timeout, and provide an infrastructure for state recovery would be beneficial for nanosatellite communications.

D. THE NEED FOR CYBERSECURITY IN NANOSATELLITES

Bandwidth limitations and unreliable connections are not conducive to a strong cybersecurity posture that ensures data confidentiality, integrity, and assurance. This triad is a model used to evaluate the strength of information assurance policies within a system. This model captures the three core components that profile a strong security stance. Due to the limited bandwidth of nanosatellites, the approach, “any data is better than no data” is sometimes adopted. This approach reduces the applicability of compression, encryption, and integrity checks on data being transmitted and received. The application of a stronger security posture is not new to nanosatellites, as evidenced by the integration of XTEA encryption in CSP, but few cases exist of other cybersecurity methods to safeguard the data being transmitted. The few cases surveyed demonstrate a preference to hardware and radio solutions instead of software solutions. A software solution that provides the functionality and infrastructure for a stronger cybersecurity posture would be a welcomed paradigm shift in approaching communication schemes of nanosatellites.

1. Data Usage in Nanosatellites

The amount of data transmitted by a nanosatellite is largely governed by its baud rate, lifetime, and orbit. These conditions can vary dramatically from mission to mission and design specifications of the nanosatellite. Looking at the first one hundred CubeSats in 2013, Michael Swartwout determined that the average lifetime of nanosatellites is typically less than 200 days [14]. Additionally, Selva and Krejci assume an average access window of five minutes [9]. Assuming that there are nine passes total per day on an orbit, the total window of a nanosatellite can be estimated.

Extending the duration of the orbit to a calendar year, 365 days and assuming 45 minutes of access per day at a baud rate of 9600 the total data transferred in bits can be estimated for a single year to 1.183 gigabytes. This is the total data transmitted by the satellite including the headers of protocols. Assuming that the actual payload of the data is encapsulated by AX.25, and protocols like IP/TCP, then the actual useable data is less than these 1.183 gigabytes.

2. Nanosatellite Communications Information Assurance Standards

Currently there is no clear standard for information assurance in the transmission of data from nanosatellites and CubeSats to ground stations and the current methods offer few security features [6]. This lack of standard impedes a clear and thorough assessment into their shortcomings and methods on which to improve those shortcomings. A survey into the security protocols of CubeSats shows a preference towards hardware base implementations of security in the data transmitted. Information assurance in communication systems is dictated by the cybersecurity triad: confidentiality, integrity, availability.

Confidentiality refers to the property of the system to only allow authorized users or parties to access the data. For data to be considered confidential and secure, this property must be maintained at all times even if the data is transmitted across a network or between nanosatellites and ground stations. A common method to ensure the confidentiality of data transmitted is through encryption. Encryption ensures confidentiality through hard-to-solve mathematical cryptograms, by making the solutions

to the cryptograms too complex for an adversary to solve in a reasonable amount of time, but allowing the intended and authorized parties with the correct keys to access the information.

Data integrity is a property of the system that ensures the data is not tampered with in transit, storage, or at any other time by unauthorized users or environmental noise. In the case of nanosatellites, integrity allows verification that the data transmitted and the data received between nanosatellites and ground stations are equivalent. A common mechanism to integrate this property into systems is the inclusion of a CRC on each packet of data transmitted. This checksum allows the receiver too verify if the data was altered at any time between transmission and receipt.

Availability is the property of the system that ensures data is available when requested. Consuming an excessive amount of system resources can create a denial of service situation where authorized users cannot access the information. Exhaustion of memory, bandwidth, processing power, and signal interference are all mechanisms that can be used to affect the availability of information between nanosatellites and ground stations.

Information assurance in nanosatellites largely focuses on the confidentiality properties of the communication system. Integrity is easily achieved in the data stream by including a CRC on each packet transmitted, while availability impacted through FM interference is out of the scope of this thesis. To this end, information assurance is reduced to confidentiality, specifically the impact encryption has on the ease of transmission. It should be noted that confidentiality does play a small role in the integrity and availability of data transmitted. If a large object is encrypted successfully, but takes a long time to transmit, while the integrity of each transmitted and received packet may be easily verified, neither integrity or validity of the data within the object can be verified until the whole object is received and decrypted. This could lead to a situation where the bandwidth is exhausted by the data transmission only to result in poor or useless data and a waste of limited resources. In another scenario, if the encrypted data is only partially received and the nanosatellite contact window ends, while each packet can be checked for integrity there is no way to ascertain the validity of the data being received until all of the

object is received. Because of these limitations, a protocol that encrypts a stream of independent bytes, rather than the object as a whole would be preferable. Such a protocol would allow the constant decryption of data as it is being received and allow for data checks to be carried out on partial and incomplete data.

3. Nanosatellite Communications Information Assurance Assessment

A survey of information assurance systems in nanosatellites and CubeSats would be inconsistent and infeasible due to the various protocols carried out by the hundreds of satellites, and due to the small sample size of actual documented implementations of information assurance protocols. As described above, integrity and availability mechanisms can be easily surveyed in protocols like TCP and CSP, as they all account for packet repeatability and support checksums, but their approach to confidentiality through encryption is not as clear cut. The approach to confidentiality is further complicated through the addition of hardware based confidentiality instead of software based mechanisms. A survey into CSP, CubeSec and GndSec, and the MEROPE CubeSat system illustrates the challenges of implementing confidentiality mechanisms into nanosatellites and provides a measure with which to evaluate the performance of other protocols and mechanisms [2], [6], [15].

CSP is designed to support the XTEA encryption algorithm. XTEA was introduced by the TEA designers David Wheeler and Roger Needham as a solution to correct two weaknesses in TEA [16]. Like its predecessor, XTEA is designed to be minimal while still providing a high level of confidentiality on information. It is a block cipher with a block size of 64-bits and a key size of 128-bits [16]. In CSP the keys are shared before the launch of the system and can be updated by using the previous keys to exchange a new key. CSP headers have a flag for packets encryption, with no other cryptographic information being exchanged. This allows for data packets to be encrypted and secure within a with a difficult to crack key, but several attacks are documented against XTEA that would break the confidentiality of the data stream [17], [18]. XTEA encryption is based on the number of rounds used to encrypt the plaintext, increased rounds provide stronger security but come at an increased

computational cost. This computational cost makes XTEA deceivingly small, as the level of security is entirely dependent on the computational power as denoted by the number of rounds undertaken to produce the cipher text. Another detriment to XTEA is the size of the block. As a block cipher, it must use blocks of a predetermined size in its algorithm. At 64 bits, or eight bytes, this is a large block, especially if the packet sizes of each data packet is small. In the event that a one byte segment of information needs to get encrypted, that means the block would have to be padded with seven bytes of null information. The addition of these blocks could potentially increase the size of the data transmitted in an already limited bandwidth environment. XTEA in CSP operates as a cipher in counter mode [5]. In this mode each block is encrypted independent of one another through a series of exclusive logical OR functions (XORs) and summation to keep a successive counter of blocks successfully encrypted. This allows for the parallelization of encryption for faster encryption schemes, but it comes at a cost in system memory and processing power. In error propagation, if a cipher is run and the cipher text is downloaded without integrity check, XTEA in counter mode does guarantee that the error propagation ratio between cipher text and plain text is one. This means for every byte affected in the cipher text, only the corresponding byte in the plain text will be affected upon decryption [19]. This is a valuable feature for an encryption scheme that has to operate under very noisy conditions, and give XTEA a preference over other encryption mechanisms that propagate the errors during encryption to two or more blocks [19]. In 2004, Ko et al. published a vulnerability of XTEA that could lead to a complete compromise of XTEA in data that has undergone 27 rounds of XTEA [18]. This vulnerability would allow the use of related keys and differential analysis of the encryption mechanism on 27 rounds of XTEA with a success rate of 96.9% [18]. To circumvent this vulnerability, XTEA would require more than 27 rounds, and thus significantly increase the associated processing cost of confidentiality. In 2009, Lu presented a related-key rectangle attack on 36 rounds of XTEA [17]. This attack, much like Ko et al.’s attack, would require an increased number of rounds in XTEA to ensure confidentiality. This is a tremendous burden for a low power system onboard a nanosatellite that could leave transmitted data vulnerable.

Developed by Challa et al., the CubeSec and GndSec security solution is described by its developers as “very light-weight” and provides authentication, confidentiality, and integrity through the use of symmetric pre-shared keys [6].The proposed solution by the authors uses Advanced Encryption Standard (AES) and Data Encryption Standard (3DES) in Galois/Counter Mode (GCM) and is implemented through hardware [6]. The reason for hardware implementation of these block ciphers is due to the high processing and time cost associated with AES and DES hardware, which the authors document in [6]. Using microcontrollers to encrypt the data and spare the processor from computing power is a resource efficient approach, but still comes with some associated costs. While methods like XTEA are directly measured in computing resources, the CubeSec and GndSec mechanism’s cost is in weight and volume on the spacecraft. The authors profile the encryption hardware with a footprint of approximately 5cm by 5cm and a total weight of approximately 9.6 grams [6]. While this footprint may seem trivial in larger spacecraft, the authors also recommend a redundant backup system that effectively doubles this physical footprint and can be a serious detriment to nanosatellites [6]. Additionally, the authors do not discuss the financial costs of the additional hardware, which should be taken into consideration given that the hardware is not recoverable after a mission. Some of the advantages offered by this system is the strong implementation of security through AES and 3DES operating at 128 bits. Additionally, much like XTEA in counter mode, GCM allows for parallelization of encryption, resulting in much higher encryption rates, while keeping the encryption costs within the hardware implementation and not severely impacting the power consumption of the spacecraft as a whole. Overall the CubeSec and GndSec system provides a valuable solution to information assurance, but at a cost in space, weight, and system complexity that may keep it out of reach from institutions.

An interesting case is the Montana EaRth Orbiting Pico Explorer (MEROPE) CubeSat built by the Space Science and Engineering Laboratory at Montana State University [15]. This project highlights the need for COTS subsystem designs in CubeSats to mitigate the lack of expertise in CubeSat design teams. The communication subsystem design goal was to have a device with a low volume profile that communicates

using the AX.25 protocol [15]. Analyzing the design and performance specifications described by the authors, it is clear that the MEROPE CubeSat did not have a mechanism to provide confidentiality to the data it was transmitting. The lack of such a protection and goal to utilize a COTS MEROPE communication subsystem indicate a serious vulnerability in the design of the MEROPE and in other CubeSats: most teams lack a network design and information assurance specialist. While the MEROPE team was well versed in the design and application of AX.25 protocols and was able to build the communication subsystems, they acknowledge their lack of technical expertise and the driving factor it was in the selection of their COTS communication subsystem. This assessment indicates the vulnerability of not only MEROPE, but also of other CubeSats. The community seems to lack a clear information assurance standard which could be explained by a lack of information assurance professionals actively involved in the development of the satellites.

Overall the survey of these systems indicates a serious need for information assurance standards that provides a high degree of confidentiality. While no system implementation comes without a cost, designing a protocol that minimizes the costs of current systems would be an asset to the community. Such a protocol would require the participation of information assurance professionals and nanosatellite designers to ensure a high degree of information assurance, keep within the operational parameters of designers, and maintain the functionality provided by other more data expensive protocols. Such a solution could provide an open source flexible standard that can be used by any design team regardless of technical expertise.

E. ENCRYPTION AND ONE-TIME-PADS

Encryption provides information assurance to data stream through cryptography. The strength of encryption varies between encryption mechanisms and the many modes they run on. Some encryption mechanisms provide stronger encryption, making them hard to crack but come at a large cost in memory and processing power, while others are light weight but have vulnerabilities. The strength of the encryption mechanism is typically measured by the ability of the adversary or unauthorized party to decipher the

data being stored within a reasonable amount of time. As processing power continues to increase, the strength of these mechanisms falters, and stronger, more computationally expensive systems are required. There are encryption mechanisms that are classified as "perfectly secure" that can be implemented easily. Mechanisms are defined as perfectly secret as an encryption scheme if the cipher text reveals nothing about the plain text, and that a given cipher text can be translated into any plain text of equal length to the cipher text with all possibilities equally mathematically probable [20]. A one-time pad (OTP) is such a mechanism.

1. Evaluating the Strengths of One-Time Pads

OTPs are, as described above, perfectly secret. This means that a string of length *n* when encrypted with a OTP of the same length, produces a cipher text of equal length. If an adversary intercepts a cipher text encrypted with a OTP, then assuming the message is limited to capital alphabetic characters, any combination of letters is equally probable due to the fact that the information that is utilized in the OTP is completely random (Figure 2).

Message	OTP	Cipher Text	Possible Solutions
"ATTACK"	"123456"	"STMOPC"	"GIVEUP"
			"LETYOU"
			"DESERT"

Figure 2. One-time pad example on alphabetic message of length 6 and a few possible solutions

OTPs are also efficient methods of encryption as each byte of information is encrypted only once in an XOR operation. This eliminates the need for multiple passes to ensure a high level of confidentiality, at a low processing cost. Furthermore, unlike block ciphers with fixed block sizes that result in padding of data and extra data being sent, OTPs do not alter the length of the message being sent. These properties arise from the fact that OTP encryption encrypts each byte individually and independently from the rest

of the data [20]. This increases the encryption strength and limits the propagation of errors as each affected byte in cipher text will only affect the corresponding plain text byte upon decryption. OTPs are currently the strongest method of encryption since all possibilities are equally probable.

2. Limitations of One-Time Pads

OTPs must meet certain criteria to ensure their perfect secrecy. The constraints limit their proliferation into practical uses. In 1919, Gilbert S. Vernam was awarded a patent for an encryption mechanism using a OTP and the XOR operation [21]. This system would encrypt a message with a OTP stored in a punch tape stored in a loop, which was later revealed to be vulnerability. By storing the OTP in a loop and reusing the key, cryptanalysis was possible as the key and character combinations were bound to be repeated in a cyclical manner, allowing adversaries to crack OTP encryption in Vernam's device [22]. In order to mitigate this vulnerability, the OTP key must be non-repeating or reusable and must also be truly random. These two criteria must be true for the entirety of the OTP, meaning the OTP must be at least as long as the total data transmitted through the mechanism. This drawback prevents the practical implementation of a prolonged use of OTP for the transmission of large volumes of data, as this rapidly increases the required size of the OTP. Another detriment of using the OTP for the transmission of large amounts of data is the need for the OTP to be truly random. If a pseudo-random number generator is used, like the large portion of random number generators in computer system, the adversary may be able to correctly deduce the pseudo random number generator and seed. This would result in the adversary being able to predict and effectively break the OTP encryption of the data. Truly random numbers can be generated through entropic processes such as radioactive decay or quantum events, and can be difficult to generate. This can be mitigated with large repositories of existing random numbers, but this presents the opportunity for an adversary to deduce which repositories are being used. Another challenge for OTP usage is the need to exchange the OTPs with the keys between the users. Asymmetric encryption mechanisms allow for the establishment of secure tunnels so that keys can be exchanged and create tunnels of information that are encrypted through symmetric keys. OTP transmissions would either

still require asymmetric key mechanisms and a large transfer of data for the contents of the OTP, or some physical exchange of OTPs. This presents a challenge since the nodes transmitting and receiving may not all be physically accessible.

These are a few of the limitations of using a OTP as an encryption mechanism. Despite its strength and perfection, practical limitations make the deployment of OTP encryption mechanisms, especially in larger data transfers as we see on the internet today. There are mitigation techniques to overcome the limitations of OTPs, such as the availability of large storage disks and large repositories of quantum information. Exchanging keys, presents a physical problem that can be avoided if the original OTP was large enough to accommodate the total lifetime data transmission and the keys were exchanged once. Overall, OTPs are a strong, albeit slightly impractical, encryption mechanism that compensates for their logistical hurdles through the level of security they provide.

3. One-Time Pads in Nanosatellites

Nanosatellites are prime candidates for the implementation of a OTP encryption mechanism. Their design and operation conditions are ideal for OTPs, and such a mechanism would provide the security needed by the spacecraft.

Several of the drawbacks of using a OTP presented above, can be effectively mitigated just through normal satellite operations. A OTP remains valid as long as the OTPs used by the receiver and transmitter are kept secret. In the case of nanosatellites, the vulnerability of an adversary obtaining the OTP is drastically reduced as one of the OTPs will be in LEO. Another shortcoming is the need to have a OTP be as long as the total data transmitted throughout the life of the mechanism if key exchanges are to be avoided and the OTP must be full of truly random numbers. As discussed in Section D.1, the total data usage of a nanosatellite in a year can be estimated to be about 1.183 gigabytes. Even if a nanosatellite mission has a lifetime of several years, data storage is currently compact enough in solid state media that a device storing a large OTP would not be a problem. In the case for the need of truly random data, several universities provide free open repositories of terabytes of quantum data to be used as random data.

This can mitigate the need to build a mechanism to generate the data, especially if it such repository data can be made private.

The drawbacks of OTP encryption make it impractical for use in large transfers of data over large networks such as the internet. Point-to-point communication between a ground station and a nanosatellite with limited bandwidth and total lifetime data transfers present ideal candidates for the implementation of a OTP encryption mechanism. These mechanisms provide perfect secrecy, a high level of confidentiality, are lightweight, and can

F. CHAPTER SUMMARY

A survey into the current state of CubeSat and nanosatellite communications, demonstrated the need for information assurance standards, and the need for lighter protocols due to the limited bandwidth of the devices. Designing a lightweight protocol for use with nanosatellites has to take into consideration the large number of constraints in data transfer rates, error rates, and processing and transmitting power available to the spacecraft while keeping in mind the design and data transfer needs of the designers. Mechanisms like IP/TCP provide the functionality at a high overhead cost, while encryption mechanisms for information assurance are a constant balance between weight, power, and processing costs. Nanosatellites provide a unique opportunity to establish a new protocol for low bandwidth communications that provides the necessary functionality and that integrates the infrastructure needed for an encryption mechanism based on a OTP. Communicating at 9600 baud over UHF and VHF is an error prone, slow connection that is currently without a clear standard. To remediate this the Nanosatellite Encrypted Reliable Datagram Protocol (NERDP) is proposed.

THIS PAGE INTENTIONALLY LEFT BLANK

III. ENCRYPTION MECHANISM

A. INTRODUCTION

Nanosatellites have limited processing power and as such require lighter encryption schemes. Protocols like CSP use XTEA, but still require multiple rounds of encryption to ensure that the encryption is strong. To mitigate this, the proposed NERDP encryption is reliant on a practical implementation of a one-time pad (OTP). This mechanism ensures perfect secrecy, and results in a strong encryption of the file at a low processing cost. The requirements for our encryption scheme are informed by discussions with the NPS SSAG (Naval Postgraduate School Space Systems Academic Group).

A key approach to the design of the mechanism is to treat the encryption and decryption scheme as a modular addition to the NERDP. By doing so, it allows greater flexibility in the implementation of the mechanism and allows NERDP to be a standalone protocol that operates even without encryption. This approach allows the independent development of the two mechanisms, and the NERDP more flexible should it be used in conjunction other encryption schemes.

B. GOALS OF ENCRYPTION MECHANISM

The encryption mechanism functionality of NERDP is specifically designed to take into account the limitations of nanosatellites and small satellites. To ensure the development of the protocol aligned with the needs of nanosatellite designers, the encryption mechanism needs to balance several attributes and performance factors while still being a feasible alternative to current encryption mechanisms.

The encryption mechanism needs to be lightweight in its processing performance to accommodate the various types of satellites that will use NERDP. To this end, one of the goals is the minimization of rounds or iterations needed to encrypt the file. By reducing the number of rounds and iterations, the processing cost is reduced and reduces the minimum processing power needed by the hardware. The mechanism should minimize the operations needed to carry out the encryption itself and should work with basic operations. This minimization of steps within the actual encryption of the data and

the use of only basic mathematical operations, such as XOR, reduces not only the processing power and time needed by the encryption mechanism, but also reduces the overall size and complexity of implementing the encryption scheme in the operation of the nanosatellite communications package. Finally, the size of any supporting key infrastructure such as keys or certificates should be minimized.

Finding a balance of these three key goals is crucial in designing the implementation of the encryption mechanism. To simplify the design of the mechanism, these goals were prioritized from most to least important as follows:

1. Number of iterations and processing
2. Complexity and number of operations per iteration
3. Size of supporting infrastructure

The reasoning behind this prioritization is the realization that memory and storage space are much less expensive in cost and data volume. By reducing the cost of the supporting infrastructure, in this case the large size of the OTP required to encrypt all of the data transmitted throughout the lifetime of the spacecraft, design of the encryption mechanism can focus on reducing iterations and complexity and their impact on processing and power consumption.

C. DEVELOPMENT OF ENCRYPTION MECHANISM

The focus on the goals of the encryption mechanism and the target community requirements facilitated the development of the mechanism. This development was carried out in a virtual environment to better measure its performance and to observe the data being encrypted. Full development of the virtual environment mechanism testbed can be found in Appendix A.

1. Mechanism Development and Platform

The platform for mechanism development needs to emulate nanosatellite functionality. Since most nanosatellites, utilize COTS components, development is emulated COTS software and operating systems and can be scaled down to more

appropriate hardware if needed. To emulate this readily available COTS software, a virtual machine was run on an Alienware Area 51 PC operating a 64-bit Windows 10 Home, an x64 based Intel Core i7-6800K CPU at 3.40GHz processor, 16 gigabytes of memory, and 1 terabyte of hard disk space as shown in Table 1.

Table 1. Host system specifications for development platform

Hardware	Alienware Area 51
Operating System	Windows 10 Home (64-bit)
Architecture	x64
Processor	Intel Core i7-6800K at 3.4GHz
Memory	16 Gigabytes
Hard Disk	1 Terabyte

The virtual machine hypervisor selected was Oracle VirtualBox version 5.1.16, and hosted a Linux virtual machine running Ubuntu 16.04 LTS, at 2 gigabytes of available memory, 16 gigabytes of available hard disk space, and utilizing 1 core of the host machine processor. The mechanism was written to operate on Python 3.5.2 in the Linux virtual machine, and written on the host Windows machine on Sublime Text Editor 3 Build 3126 as depicted in Table 2.

Table 2. Hypervisor and virtual machine specifications for platform

Hypervisor	Oracle VirtualBox Ver. 5.1.16
Operating System	Ubuntu 16.04 LTS
Memory	2 Gigabytes
Hard Disk	16 Gigabytes
Programming Language for Platform	Python 3.5.2
Text Editor	Sublime Text Editor 3 Build 3126 on Windows 10

This setup allows a quick development and testing of the platform and encryption mechanism. By utilizing Python, but not using any external libraries or dependencies, the development of the platform can be modeled in other languages with relative ease since

one of the goals of the platform is also the utilization of basic logical operators. Since OTP encryption is largely dependent on the use of XOR to encrypt the data, Python allows a user-friendly environment that allows functions also found in other languages like x86 NASM Assembly [23].

2. Mechanism Design and Operation

Setting up the testing platform and environment allows the encryption mechanism to be written in Python and tested in Linux. The design utilizes a pre-written message and OTP and carry out an XOR operation between the message and the corresponding OTP. The message then is written to a file, read from the file, and decrypted by carrying out an XOR operation with the OTP of the simulated receiver (Figure 3). Since both OTPs have been pre-shared and are being read from the same buffer, the writing to a file and reading from a file is utilized to simulate the data transmission (Figure 4).

Original Message		“0101 1100”
One-Time Pad	XOR ⊕	“1100 0111”
Encrypted Message		“1001 1011”
Encrypted Message		“0101 1100”
One-Time Pad	XOR ⊕	“1100 0111”
Decrypted Message		“0101 1100”
Original Message		“0101 1100”

Figure 3. OTP encryption utilizing logical exclusive or (XOR) function on a single byte

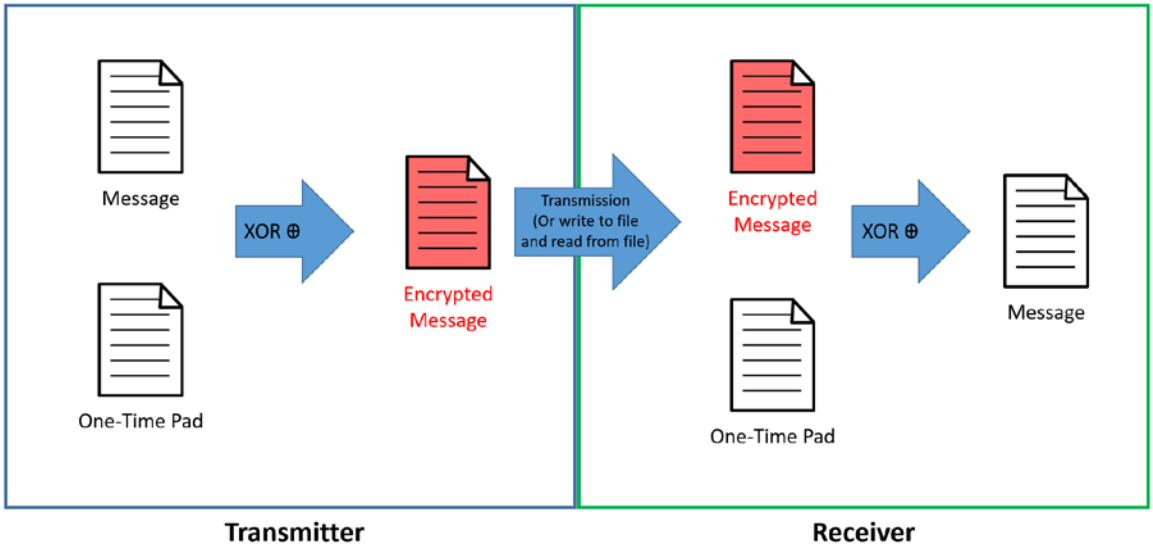


Figure 4. OTP encryption scheme on an entire message

Development largely focused on encrypting 75 unique ASCII characters comprised of all alphanumeric characters, common punctuation and symbols, and the NULL character stored in a variable called “string” (Figure 7). Note that this data is only used due to the ease of visual representation of data, when in reality any value that can be stored in a byte should be equally capable of being encrypted by the mechanism without any alteration.

```
242     # Set up the data string we want to encrypt (almost all ascii characters in this case)
243     string = '\x00abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890!@#$%^&*()\x00.,'
```

Figure 5. ASCII characters used to test and analyze the encryption mechanism

Development utilizes a pre-populated OTP with the same value for every character. In this case the ASCII character ‘1’ was utilized and stored in a variable titled “padLong” to later be used (Figure 6). It should be noted that this OTP, while violating the criteria for true secrecy wherein each value of the OTP must be a random value, still provides a working example of OTP encryption. Once developed, this OTP can be replaced with true random values and the mechanism will achieve true secrecy with no other modifications.

```
244     # This is a test OTP for development, the padlong string must be random values for a true OTP
245     padlong = 1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
```

Figure 6. Test OTP utilized for development of mechanism

In order to achieve the encryption of the whole message in Python, the mechanism takes the data to be encrypted and converts it to an integer representation utilizing the ord() functionality. For this platform, a function was written that takes a string containing the message to be encrypted and a list containing the individual values of the OTP as its input. Python then utilizes the bitwise XOR operation and appends the message to a list called “encryptedMsg” so that it can be stored or transmitted and returns the values of this list back to the main program body (Figure 7). To decrypt the message, a function is also created that will take the data received,

```
16  def encryptMessage(message, pad):
17      # ************************************************************************
18      # Encrypt the message converting the values into int and XORing them
19      # ************************************************************************
20      encryptedMsg =[]
21      for m,p in zip(message,pad):
22          cipher = (ord(m)^ord(p))
23          encryptedMsg.append(cipher)
24      # print('inside encrypt func: ',encryptedMsg)
25      return encryptedMsg
```

Figure 7. Function developed to encrypt a message of arbitrary length with a corresponding OTP

This simple operation is all that is required to encrypt any message with a OTP and supports two of the goals initially established for developing an encryption mechanism for NERDP. By only requiring a single pass per byte to encrypt, OTP encryption dramatically reduces the number of operations needed to encrypt the data. On the receiving end, in order to decrypt the data, the receiver must take the data received into a list and carry out the same operation (Figures 3, 4). When written into Python, the decryption mechanism is nearly identical as the encryption mechanism for each byte. In decryption, each byte is again converted into its integer representation, and then converted back to its ASCII representation (Figure 8). This conversion is not necessary

under normal operation of the decryption mechanism, and can be avoided. It is introduced into the platform to aid in data collection and processing since all of the characters in the decrypted message were representable by ASCII.

```
70  def decryptMessage(unpackedencryptedMessageList,pad):
71      # ************************************************************
72      # same as encrypt
73      # ************************************************************
74      decryptedMessage = []
75      for e,p in zip(unpackedencryptedMessageList,pad):
76          clearText = chr(e ^ ord(p))
77          decryptedMessage.append(clearText)
78      return decryptedMessage
79
```

Figure 8. Function developed to decrypt a message of arbitrary length with a corresponding OTP

Overall the goal of the encryption and decryption mechanism was simple and straightforward. Utilizing logical bitwise operator XOR, the platform developed illustrates the lightweight properties of the encryption mechanism utilizing a OTP. A critical consideration is that the message transmitted during development is only 75 bytes long and thus the OTP is also 75 bytes. Under normal operation, the OTP would be the same size as the total data sent over the lifetime of the nanosatellite. This operation would also require the ability to read and establish an offset from which the OTP would be read by the mechanism from the OTP preloaded file. While these mechanisms add complexity and require additional hard disk space, the complexity is mitigated since all encryption mechanisms require input to be read from the file being encrypted and additional space for supporting infrastructure is favorable over increased costs in processing and complexity.

D. EVALUATING MECHANISM PERFORMANCE

Encryption and decryption mechanism development was driven largely by performance metrics and the goals established for the design. By developing in Linux and Python, the mechanism can be ported to other platforms with ease and its performance

can be relatively consistent throughout. To better evaluate the mechanism, several key performance metrics were decided upon and development was designed around them.

1. Data Sizes of Encrypted Files

Treating the encryption mechanism as a modular addition to NERDP allows development to focus on encryption of data as independent of the packet structure of NERDP. This allows for entire files of data to be encrypted and stored without being dependent on the behavior of NERDP. While the encryption mechanism complexity and performance has already been established as lightweight despite the large space needed to store the OTP, performance of the encryption mechanism focuses on analyzing the sizes of the original files and the encrypted counterparts.

The goal of encryption is to increase the confidentiality of the data being transmitted at a cost of processing power, time, and space to store the encrypted data. Some ciphers can be used in line with the data transmitters and encrypt data as it is packed and transmitted. The OTP encryption mechanism designed for NERDP reads segments of data from the file and the OTP, encrypts them, and then transmits them. This allows for reduced memory costs as a full encrypted copy of the object being transmitted does not need to be stored by the transmitter. Additionally, if NERDP is implemented without OTP encryption, each data packet can exclude the encryption step or carry out a different type of encryption without significantly altering the protocol. Having established the low processing power and time costs for OTP, the data sizes before and after encryption are utilized as a performance measure to ensure no undue memory burden is placed to store the encrypted data. Due to its operation and design, OTP encryption does not alter the length or size of an object being encrypted.

While it is possible to pad the data to further obfuscate the length of the message being sent to a designated length, this is not necessary and provides an advantage of block ciphers which require data be padded to a multiple of the bit size of the block needed [19]. This advantage supports the selection of OTP encryption as a feasible encryption mechanism for bandwidth limited operations where file sizes and data transferred are sought to be kept at a minimum.

2. Processing and Iterations

As discussed in the design of the mechanism, OTP encryption provides a lightweight solution for encryption. By requiring a single iteration and operation of the file being encrypted, the mechanism leaves a very small system footprint on the performance of the communication system as a whole. Reduced iterations and processing time, in turn result in less lag and delays when the encryption mechanism is used to encrypt data packets as they are being streamed. Further comparison of OTP encryption to XTEA encryption in Python on a Linux environment is discussed in Chapter V, Results and Analysis.

Overall, OTP encryption provides a lightweight option for encrypting data. This mechanism has low impact on processing power consumption and can be easily integrated to transmit data quickly in NERDP. These benefits come at a cost in disk space, as several gigabytes of data must be stored on the spacecraft to ensure the perfect secrecy of the transmission. Due to modern data storage capabilities, these costs are easily mitigated and the benefits far outweigh them.

E. ROBUSTNESS TO ERROR IN TRANSMISSION

Due to the low power conditions over UHF and VHF in which nanosatellites operate, the signal quality can be impacted by the introduction of errors in to the data stream. These errors can vary in severity and frequency, and can have an impact on the data being transmitted. If the error rate is too high, then the data may be useless. In order to assess the robustness to error of OTP encrypted data, its behavior under several types of error is predicted, evaluated, and ultimately simulated under the assumption that there is no error correction mechanism or data integrity requirement.

1. Insertion and Deletion of Data

Assuming that a data packet of a given length is properly read, encrypted, and transmitted by the nanosatellite, it is possible for the data to arrive either incomplete or with random noise inserted into the packet. This types of error are significant because they produce a shift in the data and can potentially affect an entire packet. If the receiver

is only expecting *n* bits from the transmitter, and it receives *n* + *x* bits, where x is the number of bits inserted, then the receiver will truncate the data received at *n* bits and *x* bits of data will be lost. This will also affect the data "downstream" of the site of insertion as each bit will now be shifted and can affect the value of all the subsequent bytes and thus the encrypted message sent within this packet. Conversely, if a transmission loses *x* bits in the packet, a similar effect occurs as all bits are shifted to the start of the message and the data is incomplete (Figure 9). Such an error occurs when the connection is not stable and data is lost or dropped in transmission.

Original Data	0101 0001 1010 1011 1111 0101 …
Insertion of 1 bit	0101 1000 1101 0101 1111 1010 1 …
Deletion of 1 bit	0100 0011 0101 0111 1110 101…

Figure 9. Inserting or deleting a single bit in the first byte propagates throughout all subsequent data until the end of the packet.

Either of these errors can have disastrous consequences for data encrypted with a OTP. Since each byte is encrypted independently with its corresponding byte from a OTP, if all of the bits are shifted from either insertion or deletion, it is possible that large portions of the entire data packet sent become indecipherable by the OTP. Additionally, since either of these errors can occur at any given time, and can occur multiple times, it is possible to impact entire packets and lose large segments of data without being able to recover any portion of that data.

2. Replacement of Data

A more common error, and the error defined by the BER, is the replacement of data during transmission. Assuming that a data packet of a given length is properly read, encrypted and transmitted by the nanosatellite, it is possible for noise and interference to alter the values of the existing data at random points in transmission. These errors known as "bit flips" will alter the data of the transmission and change the value of bits. These errors will either affect a single bit or can also occur in bursts affecting several bits at a

time. While individual bit flips only affect the byte of encrypted data containing the flipped bit, burst errors can occur at any given time. If a burst error occurs where several bits are flipped at the start or beginning of a byte, it is possible that multiple bytes are affected as the error "spills" over into the next byte (Figure 10).

Original Data	0101 0001 1010 1011 1111 0101 …
Single Bit Flip	0101 0000 1010 1011 1111 0101 …
Burst Bit Flip (2+ Bits)	0101 0000 0110 1011 1111 0101 …

Figure 10. Replacing one or more bits can have a varying degree of impact on the data, but effects do not propagate to subsequent data

Fortunately, this type of error does not alter the overall length of the encrypted data packet being transmitted and each bit after the error is unaffected by the replacement of the data. This type of error only affects individual bytes that happen to be impacted by the bit flips. In the event of a noisy signal where the rate of bit flips is high, it is still possible to do some data recovery of partial information as a large portion of the data may still be intact. This benefit directly translates to the decryption of the data since each byte is encrypted independently, and only the affected cipher text bytes will alter the data in the corresponding decrypted data bytes.

The platform utilized to develop the encryption mechanism was also used to simulate these errors. Utilizing the Python script used to encrypt and decrypt data and the *numpy* and *binascii* packages, the platform was used to simulate multiple rounds of single bit and multiple bit flip errors on the encrypted data, and then decrypted to analyze its impact on the original message. These errors were introduced given a normal distribution and a given probability to simulate various rates of error. First the data was converted from its raw bytes into its binary representation by the *binascii* package and put into a list. For any given bit at position *n*, if the probability landed that it needed to be flipped, the bit would then be flipped from "1" to "0" or vice versa and a counter of bits flipped would be incremented to later ensure the probabilities are behaving as predicted. Once

every bit of the encrypted message was processed, it was converted back to its ASCII byte representation with the *binascii* package and compared to the original message received (Figure 11).

```
def bitFlipper(encryptedMsgRead):
    # ********************************************************************************************************************
    # Take random bits and flip them with a discrete probability. Used to measure error propagation in simulation
    # ********************************************************************************************************************
    # convert the packed, encrypted message into bits and make it a list
    binaryMsg =  bin(int.from_bytes(encryptedMsgRead, 'big'))
    binaryMsgList = list(binaryMsg)
    # print('Binary MSG LIst: ', binaryMsgList)
    print('Binary MSG before flip: ', binaryMsg)

    # Based on a probability, we can establish the probability of each bit getting flipped

    # 0 for no change, 1 flips the bit
    elements = [0,1]
    # 1/16 bits needs to be flipped. so probability of flipping is 0.0625
    probabilities = [0.9375, 0.0625]

    # need to keep count of bits flipped for later analysis
    bitsFlipped = 0

    # need to skip the 0, and 1st bit since they  are there for python reasons
    for i in range(2, len(binaryMsgList)):
        # get a random probability (labeled coin for coin toss though probabilities can be changed)
        coinList = (numpy.random.choice(elements,1,p=list(probabilities))).tolist()
        coin = coinList[0]
        # if the coin is 0 then go back to the top, and increase i
        if coin == 0:
            continue
        # else the coin is not zero, so we have to change 0 -> 1, and 1 ->0 in position i of binaryMsgList
        if  binaryMsgList[i] == '0':
            binaryMsgList[i] = '1'
            bitsFlipped += 1
            continue
        if  binaryMsgList[i] == '1':
            binaryMsgList[i] = '0'
            bitsFlipped += 1

    # after the bits have been flipped time to rejoin the list into a string
    binaryMsg = ''.join(binaryMsgList)
    print('binary MSG after flip: ', binaryMsg)

    # convert to ints for python because python loves ints
    binaryMsgInt = int(binaryMsg,2)

    # we then convert back to the packed char bytes that we had originally
    encryptedMsgReadFlipped = binascii.unhexlify('%x' % binaryMsgInt)

    # quick test
    # ***********************************************
    print('No flip: ',encryptedMsgRead)
    print('Flip: ', encryptedMsgReadFlipped)
    if encryptedMsgReadFlipped == encryptedMsgRead:
        print('Same')
    else:
        print('Different')
    # ***********************************************

    # return the encrypted packed message with some bits flipped and the counter.
    return encryptedMsgReadFlipped, bitsFlipped
```

Figure 11. Function used to simulate individual bit flips in the OTP encrypted data and compared to original data

In order to simulate a burst of bit flips, if a bit was determined probabilistically that it was going to be flipped, the function would also flip the subsequent two bits for a total of 3 flipped bits. This would create random bursts in the encrypted data and would then be compared to the original (Figure 12).

```
def flipSubroutine(i,binaryMsgList):
    if  binaryMsgList[i] == '0':
        binaryMsgList[i] = '1'
    else:
        binaryMsgList[i] = '0'
    return binaryMsgList[i]

def tripletBitFlipper(encryptedMsgRead):
    # ***************************************************************************************************************
    # Take random bits and flip them with a discrete probability. Used to measure error propagation in simulation
    # ***************************************************************************************************************
    # convert the packed, encrypted message into bits and make it a list
    binaryMsg =  bin(int.from_bytes(encryptedMsgRead, 'big'))
    binaryMsgList = list(binaryMsg)
    # print('Binary MSG LIst: ', binaryMsgList)
    print('Binary MSG before flip: ', binaryMsg)

    # Based on a probability, we can establish the probability of a bit getting flipped

    # 0 for no change, 1 flips the bit
    elements = [0,1]
    # 1/16 bits needs to be flipped, so probability of flipping is 0.0625
    probabilities = [0.9375, 0.0625]

    # need to keep count of bits flipped for later analysis
    bitsFlipped = 0

    # Initialize index of the list to 2
    # need to skip the 0, and 1st bit since they  are there for python reasons ('0' and 'b')
    # thats why we start at 2
    i = 2

    while (i < len(binaryMsgList)):
        # get a random probability (labeled coin for coin toss though probabilities can be changed)
        coinList = (numpy.random.choice(elements,1,p=list(probabilities))).tolist()
        coin = coinList[0]
        # if the coin is 0 then go back to the top, and increase i
        if coin == 0:
            i+=1
            continue
        # else the coin is not zero, so we have to change 0 -> 1, and 1 ->0 in position i of binaryMsgList
        else:
            # flip the first bit at position i
            binaryMsgList[i] = flipSubroutine(i,binaryMsgList)
            # increase thindex to i+1
            i+=1
            bitsFlipped+=1
            # if i+1 < EOF then we flip it and increase i to i+2
            if (i< len(binaryMsgList)):
                binaryMsgList[i] = flipSubroutine(i,binaryMsgList)
                i+=1
                bitsFlipped+=1
            # if i+1 succeded in flipping, then we test if i+2 < EOF, if not then we are done and go back to while loop which will exit
            # if i+1 failed then this will also fail and we will go back to the top of the while loop
            if (i< len(binaryMsgList)):
                binaryMsgList[i] = flipSubroutine(i,binaryMsgList)
                i+=1
                bitsFlipped+=1

    # after the bits have been flipped time to rejoin the list into a string
    binaryMsg = ''.join(binaryMsgList)
    print('binary MSG after flip: ', binaryMsg)

    # convert to ints for python because python loves ints
    binaryMsgInt = int(binaryMsg,2)

    # we then convert back to the packed char bytes that we had originally
    encryptedMsgReadFlipped = binascii.unhexlify('%x' % binaryMsgInt)

    # quick test
    # ***************************************************
    print('No flip: ',encryptedMsgRead)
    print('Flip: ', encryptedMsgReadFlipped)
    if encryptedMsgReadFlipped == encryptedMsgRead:
        print('Same')
    else:
        print('Different')
    # ***************************************************

    # return the encrypted packed message with some bits flipped and the counter.
    return encryptedMsgReadFlipped, bitsFlipped
```

Figure 12. Function used to simulate burst bit flips in OTP encrypted data and compared to original data

While there are other types of errors that can occur, according to conversations had with James Horning at the Naval Postgraduate School Space Systems Academic Group, these three types of errors are the most common and are the ones that can have the most serious impact to the effective decryption of data by the receiver. The low power combined with the already low bandwidth make errors prevalent in the datalink, so understanding the behavior of the encryption mechanism is crucial. Further results of the simulated errors are discussed in Chapter V, Results and Analysis.

F. POSSIBLE SOLUTIONS FOR ERROR PROPAGATION

While the error propagation from signal noise may affect substantial amounts of data, the error can be contained within the data packets and not affect the whole data stream. Analyzing the errors and simulating the errors in a controlled environment allows for an assessment of possible solutions to mitigate the impact of the various types of errors. While there is error correction hardware available on both the transmitting and receiving ends, the focus for the evaluation of the encryption mechanism is centered on possible software solutions integrated into either the encryption mechanism or NERDP. Most solutions center the usage of a data integrity check such as a cyclic redundancy check (CRC) to verify if an error occurred, then apply several error correction mechanisms depending on the type of error.

1. Encryption Mechanism Error Correction

At its current stage of development, the encryption mechanism does not provide any functionality in correcting error on either the transmitter or receiving end. One of the drawbacks the modularity of the encryption mechanism is the disconnect between the encryption module, the NERDP scheme, and the AX.25 (or other) protocol. Drawing from the OSI layer model, the encryption operates at the application layer while most errors occur at the physical layer. This disconnect does not mean that there are no possible solutions to introduce error correction into the encryption mechanism. Assuming that the protocol used integrates an integrity check in the form of a CRC, it is possible to develop functionality that can help detect and correct possible errors in the data.

Mechanism to narrow down the error insertion or deletion location can be introduced into the receiver. Such mechanism can subdivide the data of the packet transmitted, introduce parity bytes periodically into the data, and create larger buffers than the expected data. By introducing markers periodically throughout the data in the payload, it is possible to detect where the bit shift may have happened. If every *nth* byte is full of zeroes or ones, any shift caused by insertion or deletion has a probability of altering one or more parity bytes. If that byte is altered, then the encryption mechanism can start deleting a bit before the affected parity byte affected and seeing if all subsequent parity bits fall into alignment. Additionally, the mechanism can either shift all those bits to the left or the right to attempt to find the combination with the most parity bytes aligned. In the event of data replacement, data validation of every byte may be the best approach. If the data is expected to contain bytes of a certain type or and the data does not match, it is possible to attempt to substitute multiple likely values into the data.

The downside to correcting these data sets is the fact that the perfect secrecy of the OTP means that all combinations are theoretically equally possible. This makes the attempts to "guess" the correct combination of bits that will provide the right CRC or checksum a computationally expensive problem. Error correction in the signal from the perspective of the encryption mechanism while hypothetically possible is not realistic or feasible.

2. Data Loss and Reliability

Integrity checks are easy to implement and are computationally inexpensive. Calculating the CRC or checksum of a message provides a mechanism to validate the data received with the data sent. These integrity checks not only allow the validation of data, but can also be used to reject data sets deemed too unreliable or tainted. Due to the high levels of noise in the nanosatellite transmission signal, data loss from deletion of entire packets in the transmission and from the invalidation of packets from lack of integrity is not uncommon.

Mitigating this data loss through error correction and prevention can get computationally expensive and introduce so much complexity to a system that it defeats

its goals of being lightweight. As a solution against this, the concept of reliability is introduced as a tenet for mitigating data errors and a crucial function of NERDP. Reliability offers the ability to discard packets based on their lack of integrity and request retransmission of packets. This retransmission is used seeking that the errors accrued in the first packet no longer affect the same packet that has now been retransmitted. If the retransmitted packet also fails the integrity check, then a retransmission is requested and can be requested ad nauseam until the integrity check is passed. A possible solution to this possible infinite retransmission, is the averaging of data packets to construct a full packet out of the existing malformed packets. If both the original and retransmitted packets both fail the integrity check, the receiver can examine the differences between both of the packets and attempt to combine several permutations of the differences in the packets and calculate the integrity of these packets. If after *n* tries, the packet still has not passed the integrity check, a retransmission can be requested and each bit can be compared to the same bit in other packets. This surveying will determine what value of the bit is the most common in the other packets and will select that value to use in its recalculation of the CRC or checksum. This approach could reduce the number of retransmissions as each retransmission makes the sampled message more likely to pass the integrity check. Unfortunately, this reconstruction comes at the cost of the already limited bandwidth of the nanosatellite.

Mitigating error propagation is no small task. Integrity checks, retransmissions, and error correction are limited in their scope and can only provide so much mitigation before their costs become too high. Selecting the appropriate settings for the data being transmitted and limiting the retransmission of data to ensure that resources are well invested are the ways we can currently manage error propagation. In some cases, disregarding integrity checks, accepting incomplete data, and doing away with all encryption is the only option in these volatile environments because some data is better than no data at all.

G. CHAPTER SUMMARY

Taking into consideration the environment, operation, and implementation of nanosatellites, an encryption mechanism for NERDP was proposed. This encryption mechanism was designed to operate on any nanosatellite capable of running the most basic of software, and was demonstrated using Linux and Python in a virtual environment. This design was bounded by goals and guidelines that accurately reflected the needs of the small satellite and nanosatellite community. This encryption mechanism provides a strong information assurance posture at a low cost to processing and time. Its constraints from a data storage perspective and its vulnerability to errors were taken into consideration into its design and implementation. A thorough analysis still supports that the OTP encryption method may be the fastest most reliable encryption method for nanosatellites. Utilizing Python as a development and testing platform for the design of the encryption module supported by NERDP provides a proof of concept of the implementation of an encryption scheme and highlights the feasibility of utilizing it in future nanosatellite and small satellite missions.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. NERDP STRUCTURE AND DEVELOPMENT

A. INTRODUCTION

Nanosatellites have limited bandwidth when operating in the UHF and VHF bands. This bandwidth commonly operates at 9600 baud as described in Chapter II. This limited bandwidth, in combination with the propensity to errors in the noisy signal, results in a restrictive environment for which normal packet transfer protocols may not be well suited. A high loss of data packets due to signal noise and failure to meet integrity requirements can be mitigated in IP connections like TCP by requesting a retransmission of the data [11]. While this solution may be inexpensive in connections operating at a million-bits-per-second data transfer rate, in the limited bandwidth environment of nanosatellites, they begin to accrue a data overhead cost. Furthermore, in order to waste less bandwidth, some nanosatellite designers may opt to use small data packet sizes for the packets over the AX.25 protocol. While TCP can send several kilobytes of data in a single packet, making its 20 byte header comparatively small, small packet sizes of less than a hundred bytes are gravely impacted by a 20% header cost [11].

The retransmission and ability to rebuild received objects from packets in the correct order, referred to as reliability, is a crucial component of protocols like TCP and CSP. These protocols ensure that the data received is not only properly ordered but also ensure the receivers can request retransmission of specific data packets. Unfortunately, TCP is a very verbose protocol that sends an acknowledgement of each data packet and is better suited for a full-duplex structure as opposed to a half-duplex (Figure 1). Implementing reliability through IP/TCP in nanosatellites is expensive since TCP cannot operate without the IP header. To mitigate this, the NERDP protocol proposes a solution that will bypass the need for the IP header, and as long as data is transmitted and received by a Layer 2 protocol like AX.25, the receiver will be able to reliable reassemble the data.

B. GOALS FOR NERDP FUNCTIONALITY

NERDP is designed to work in a limited bandwidth environment. More specifically, it is also designed to take into account the need for small packets and large numbers of retransmissions from the spacecraft. These design criteria align with the current needs of nanosatellite and small satellite designers who are forced to use COTS protocols like TCP, but cannot afford the bandwidth to do so.

The protocol for transferring data packets, assembling them in the right order, and requesting retransmissions, needs to have a minimal footprint on the size of the data being transmitted by Layer 2 protocols like AX.25. Minimizing packet headers and extraneous packet transmissions are two key components in achieving this low footprint. In the case of TCP, every packet must be acknowledged by the receiver; this is a clear example of extraneous packet transmissions that NERDP sought to minimize to reduce the total volume of data transferred. On that same vein, each TCP and IP header utilize large amounts of data to specify the target machine IP address and port number on which the data should be sent too, along with other extraneous data for routing and networking that is not relevant to the operation and transmission of data by nanosatellites. By designing a protocol that provides key functionality with minimal components, the nanosatellite can better use its limited bandwidth and increase the throughput of the entire data transfer.

Another goal of NERDP design was to make it modular and flexible to support the various nanosatellite designs. This could mean that NERDP could serve as a feasible alternative to UDP that did not require the IP header, or be usable without encryption, reliability, or integrity. Modifying the logic at no extra data cost to the header, would allow for NERDP to be a feasible multi-tool transfer of data in bandwidth limited connections.

C. OVERVIEW OF NERDP BEHAVIOR

The base behavior of NERDP is to facilitate the segmentation of objects into small data packets before transmission by a Layer 2 mechanism. These data packets are associated into frames of 255 packets and are transmitted sequentially with intervals for

synchronization and retransmission of packets within a particular frame. Unlike TCP which requests retransmissions on a packet by packet basis, NERDP utilizes a burst retransmission request. A burst retransmission request is a single packet sent by the receiver after the transmitter has finished sending a frame. A frame in NERDP is the maximum number packets, 255 packets, that can be sent by NERDP before the transmitter switches to receive mode and awaits a request for retransmission with a retransmission packet from the receiver. This single packet in one transmission, requests all possible missing packets from the frame. Once they are received and the frame is completely downloaded, the receiver signals the transmitter to begin the transmission of the next data frame. Once the next frame begins to send the data to the receiver, the previous frame is discarded making it impossible to retransmit that data unless the whole process is restarted. This sequential transmission of frames continues until the entirety of the object has been sent

Under ideal conditions with no errors, regardless of whether or not the data is encrypted, a nanosatellite can use NERDP without any retransmission of packets from loss of data or failure to pass an integrity check.

1. Base NERDP Behavior

Typical operation of NERDP in the request of an object from a nanosatellite begins by the nanosatellite being in a perpetual “listen” state where it is constantly receiving data, and the ground station sending a request (REQ) packet to the nanosatellite over port 0 with the object name and the port number the ground station requests the data packets to be sent (Figure 13).

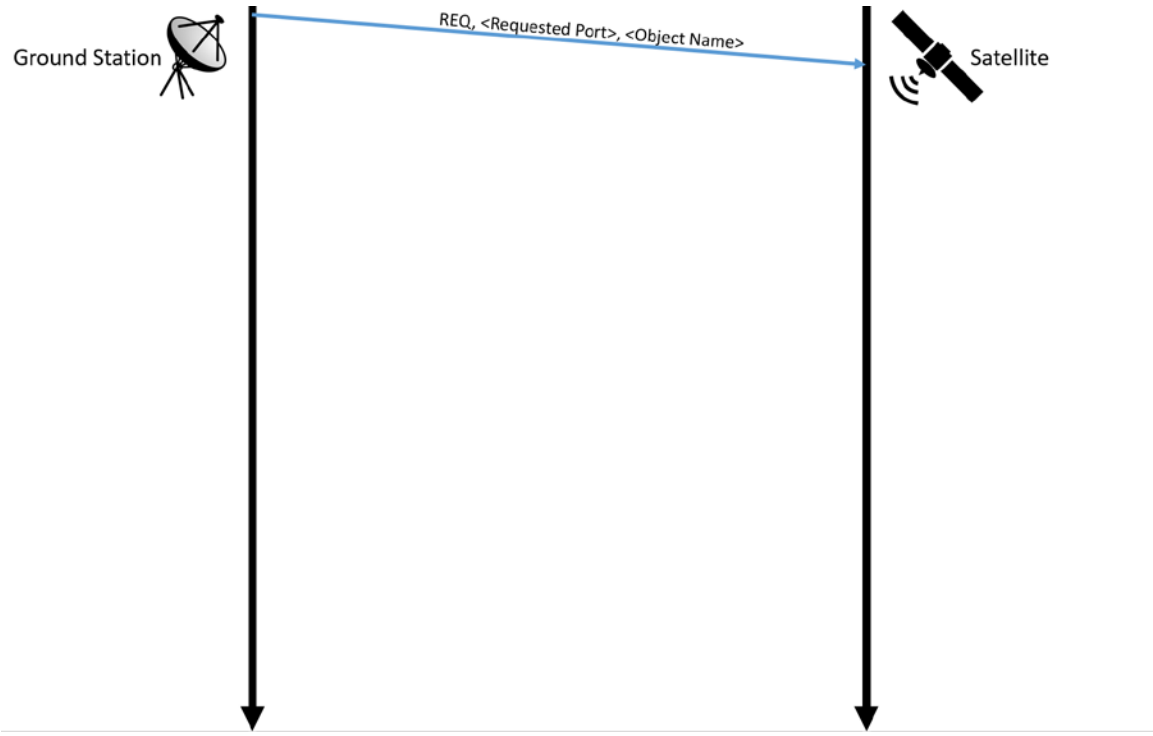


Figure 13. Requesting an object to a specific ground station port sent from port 0

NERDP operates on a system of channels for different types of data analogous to TCP and UDP port numbers. These ports redirect the different types of packets and data to different logic within the receiving and transmitting logic. NERDP packet designs currently use the 16 destination and 16 source ports for all transfer of data. These ports are specified in the packet header for each packet to route the packet to the correct parsing logic. Currently port 0 is used for all control packets for data transfer, port 1 is used for satellite state of health data, while data transfer packets are limited to ports 2 and higher.

Once the nanosatellite receives the request packet, it opens the object and sends an acknowledgement (ACK) packet to the ground station's port 0. This acknowledgement contains metadata about the data transfer such as the offset at which the satellite started reading from the OTP (if encryption is enabled) and the size of the object. This data allows the ground station to decrypt the packets with the corresponding OTP information and be able to estimate the number of packets it is expecting (Figure 14).

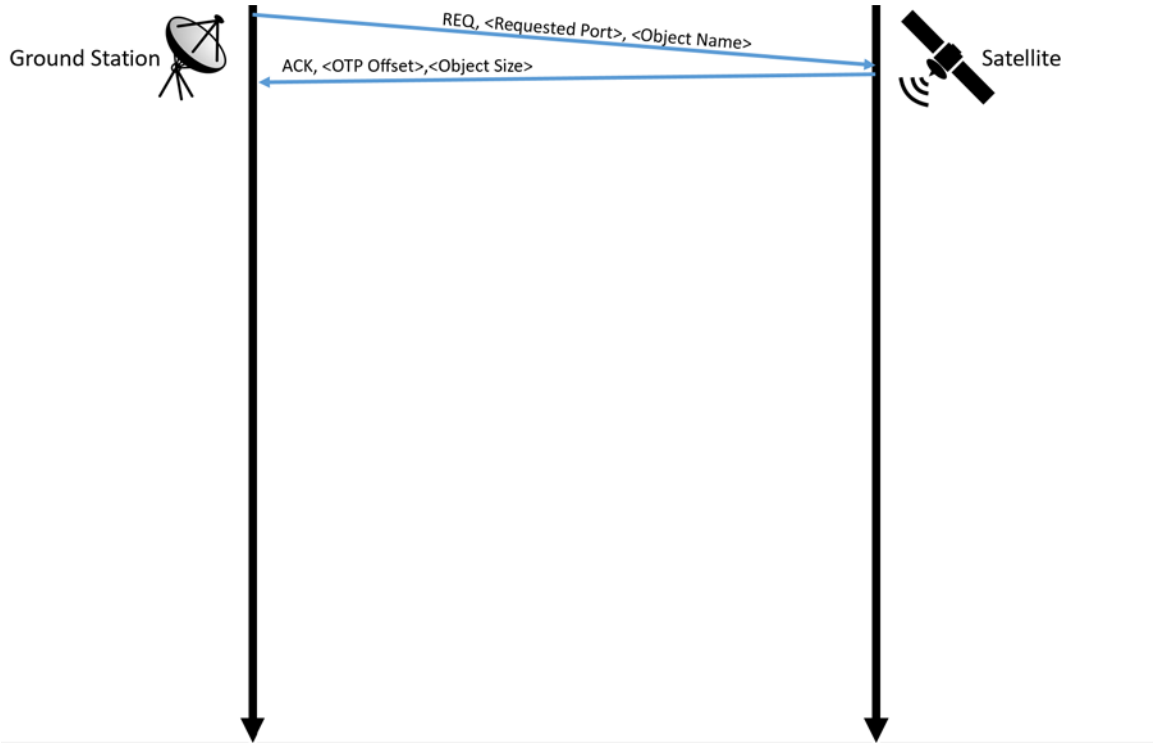


Figure 14. Acknowledgement of object request sent to ground station port 0 containing OTP offset data and object size data

Immediately after sending the acknowledgement packet, the transmitter begins sending packets of data (DAT) size *n* to the receiver's requested port, where *n* is the size of packet determined by the design of the radio. The transmitter takes *n* bytes from the object, calculates their CRC, and then encrypts it if encryption is enabled. These components are organized into data packets that are sent sequentially and received by the receiver. Each packet header also contains a packet identification number between 0 and 255, to allow NERDP to reassemble the data in the correct order. Each frame is comprised of 256 of these packets, or however many packets are needed in the last frame before the end of the object. The receiver decrypts the packet utilizing the data from the acknowledgement packet if encryption is enabled, verifies the integrity of the data and stores it in a buffer. Only after it has received all of the packets expected for that frame does it write them to a file (Figure 15). It should be noted that the final file written by the receiver contains the acknowledgement data along with the object data.

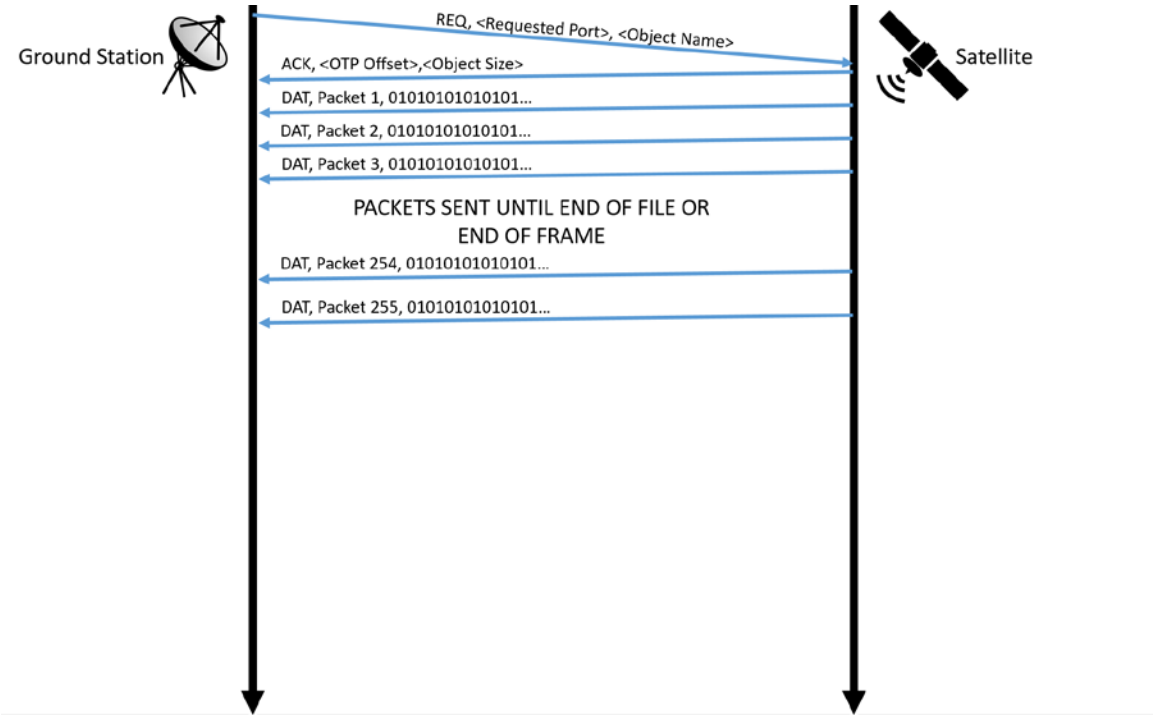


Figure 15. Transmission of an entire data frame to ground station's requested port

Immediately following the transmission of the last data packet in the frame, if the entirety of the object has not yet been sent, NERDP sends a synchronization (SYN) packet from the transmitter to the receiver requesting the burst retransmission of all packets the receiver did not receive. Under ideal situations, there is no need for retransmission and instead the receiver writes the stored packets to a file, clears the buffer, and responds with a continue (CON) packet indicating the transmitter should continue the transmission of the object with the next frame (Figure 16).

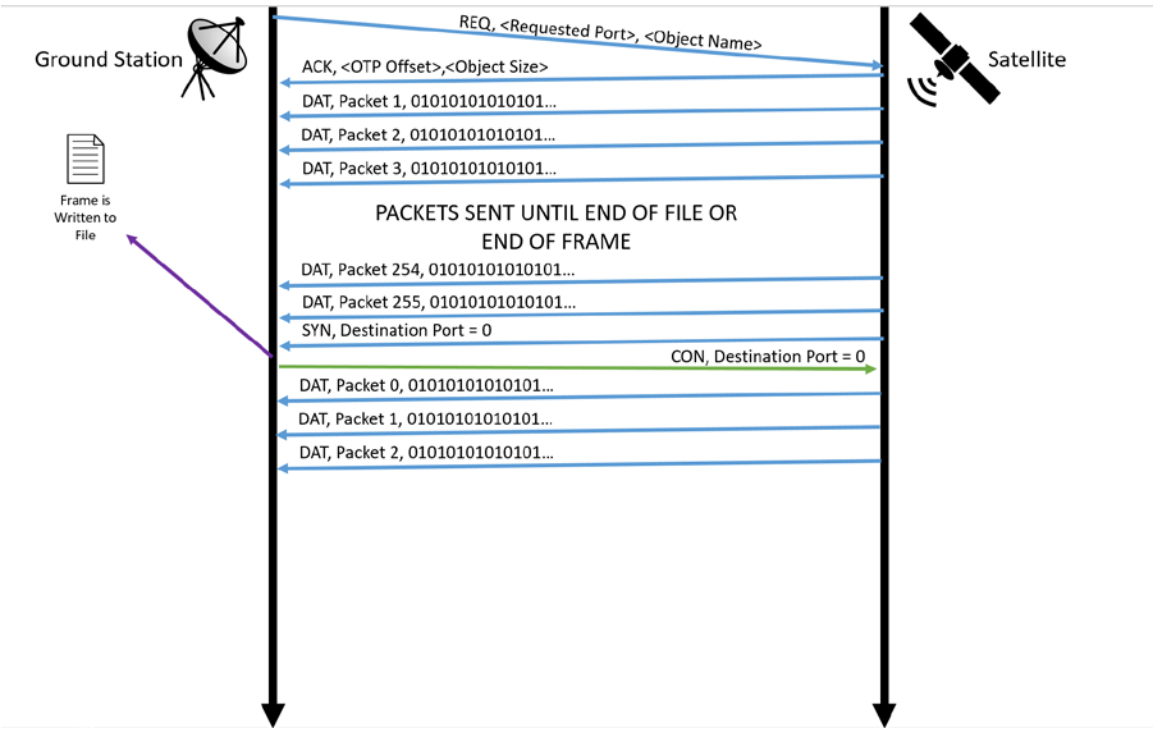


Figure 16. Synchronization and continuation of data transfer from nanosatellite to ground station using NERDP

The last frame containing data (or in the case of small files, the only frame) does not need to send the full 256 packets of information and can have any number of packets between 1 and 256. When the transmitter reaches the target size of the object it is transferring, it sends a finalization (FIN) packet instead of a synchronization packet to port 0. This packet serves as an indicator to the receiver that the entirety of the object has been transmitted. If any packets are missing, the receiver will request the retransmission of packets. Under ideal circumstances, no retransmission is required and the receiver will write the current data received in the frame to a file, and instead of a continuation packet, the receiver will also reply with a finalization packet to the transmitter's port 0. This returns the state of the transmitter to its initial state where it awaits another request (Figure 17).

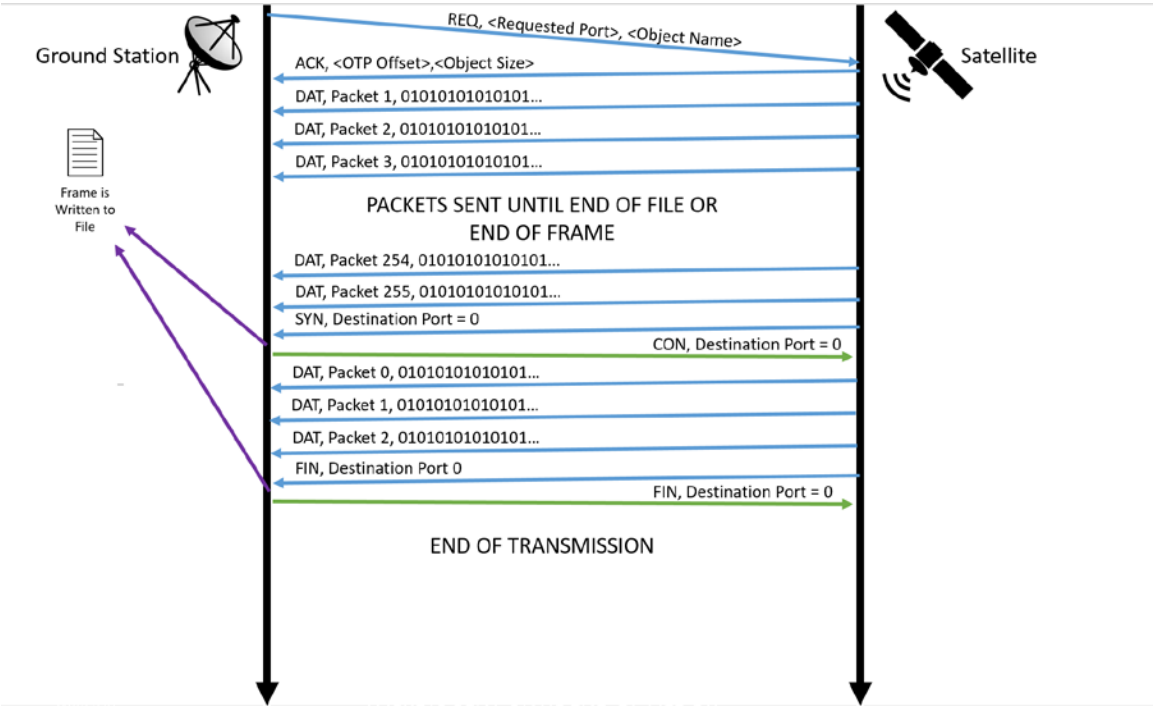


Figure 17. Finalization of data transmission with a final data frame with only 3 data packets

The base behavior of NERDP is designed with the elimination of unnecessary retransmission as a core tenet. By separating the data packet transmission into frames of packets, the need for an individual acknowledgement for each one of the received data packets is eliminated. This facilitates the use of small data packets to mitigate errors and signal noise and allows for breakpoints in the data transmission at predictable locations. Under normal operation, at any of these breakpoints, if a finalization packet is received by the transmitter instead of a request for packet retransmission or a continuation packet, the transmission of the data is immediately terminated and the spacecraft returns to its initial state awaiting a request. This function gives the ground station operator the ability to terminate the transmission early in the event that the data is corrupted or invalid. This allows for a better use of the window of time the nanosatellite is available as the ground station does not have to wait for the transfer of the entire unwanted object.

2. Reliability and Retransmission

In the event of errors in the data, NERDP has integrity built into every packet in the form of a 16-bit CRC. This CRC allows the receiver and transmitter to validate each one of the packets received. This validation ensures no corrupted data is stored. To mitigate the gaps in the data, NERDP is also designed with reliability in mind in the form of retransmissions. This functionality allows the retransmission of any packet within a given frame, and is extended to even control packets destined to port 0. If any packets are lost or corrupted in a particular frame, they can be requested again by the receiver, and put in the appropriate location in the reconstructed buffer.

Assuming normal operation allows for a ground station receiver to request an object from a nanosatellite transmitter, when one or more packets are lost, the NERDP on the ground station waits until the entire frame has been sent and a synchronization (SYN) packet has been received. It is here where the ground station, instead of continuing onto the next frame, generates a bit mask of 256 bits. These bits are all initialized at 0, and NERDP on the ground station flips the *nth* bit to 1 if the *nth* packet of the frame is missing. This generates 256 bits of 0's and 1's that can then be transmitted as 32 bytes as the payload of a missing (MIS) packets request packet. This MIS packet triggers the retransmission from the transmitter which is again immediately followed by another synchronization packet. If the receiver is now satisfied with the data, it will then write the data to file and send the transmitter a continue packet to begin the transmission of the next frame (Figure 18). If the receiver still requires additional retransmissions, missing packet requests are recalculated and sent to the transmitter and the cycle repeats until the receiver is satisfied with all of the packets in a frame.

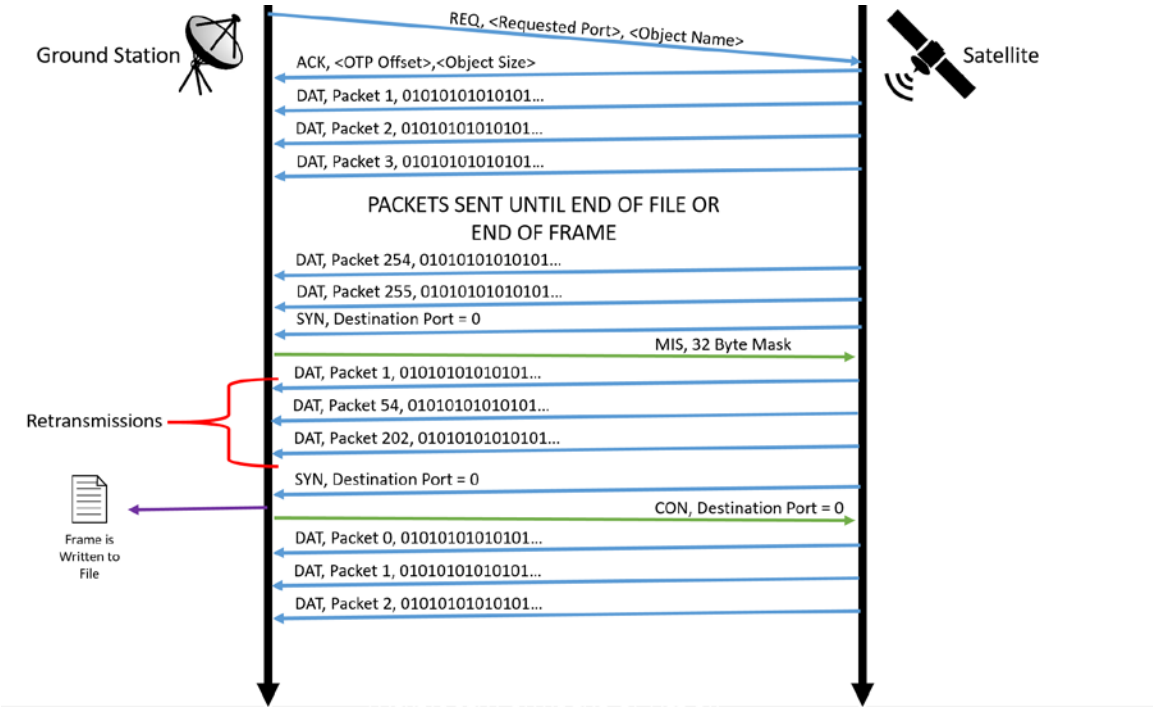


Figure 18. Retransmission of packets 1,54, and 202 utilizing the 32-byte mask in the MIS packet payload and continuing the transmission

In the event that an object requires a single frame less than 255 packets, or in the event of the last frame is less than or equal to 256 packets, the transmitter sends a finalization packet to the receiver. If the receiver requires retransmission of specific packets, it again calculates the 32-byte mask that specifies which packet is missing and sends a missing packet request to the transmitter. The calculation of this mask takes the object size received from the initial ACK packet and will calculate how many packets it expects in the final frame. If the frame requires less than 256 packets, it will pad the byte mask with 0's for all packets not expected. This makes the byte mask always equivalent to 32 bytes, regardless of the number of packets in the frame. Much like a synchronization packet, NERDP on the transmitter will retransmit the missing packets requested and send a finalization packet to the receiver. The receiver can either request retransmission of packets, or can respond with a finalization packet signaling the end of the transmission (Figure 19).

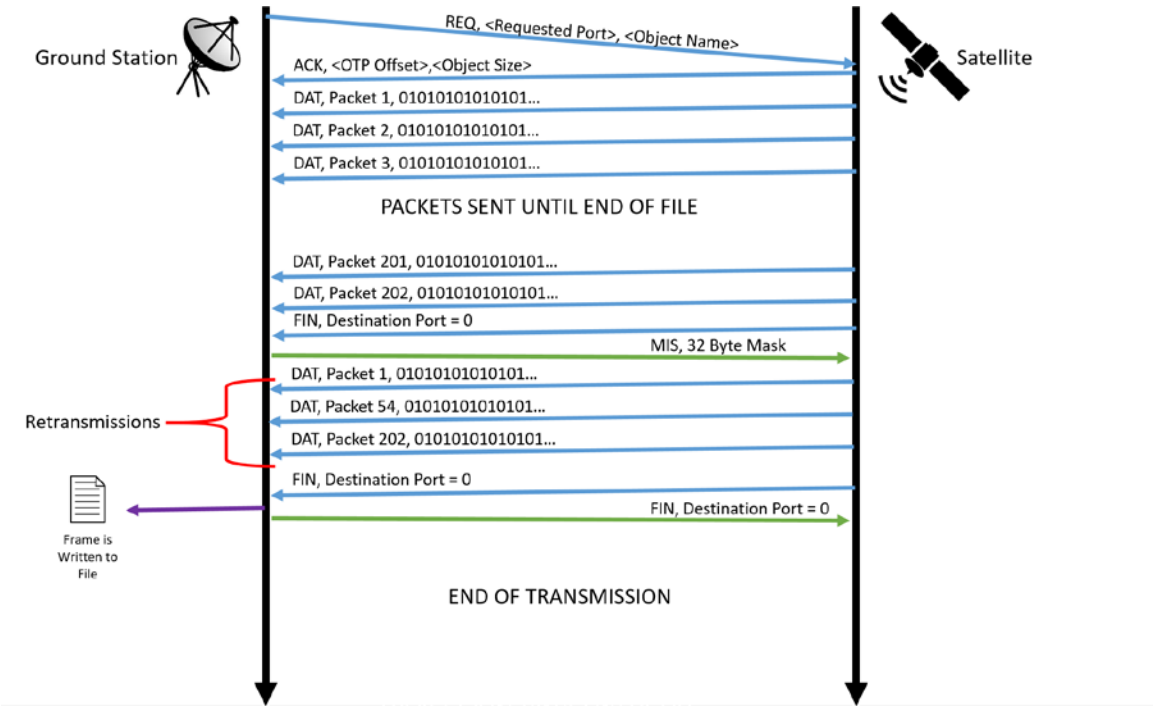


Figure 19. Retransmission of Packets in a frame containing less than 256 packets utilizing the MIS packet as a response to the FIN packet

NERDP consolidates the request for retransmission of individual packets into a single byte mask packet that allows for burst retransmission. This mechanism is good for the retransmission of data packets, and even entire frames, but it relies on the successful receipt of an acknowledgement, synchronization and finalization packets to the receiver. If any of these control packets are lost in transmission or fail the integrity check, NERDP has mechanisms to mitigate loss of control packets.

In the event of a loss of the acknowledgement packet in the first frame, NERDP on the ground station still receives and stores the other data it receives until the receipt of a synchronization or finalization packet. To mitigate the need for the object size and OTP offset in the acknowledgement packet which NERDP needs to operate successfully, upon the receipt of a synchronization or finalization packet, if the ground station NERDP sends a missing packet retransmission request to the transmitter specifically only asking for the acknowledgement packet and listens for data from the transmitter. Upon receipt of the retransmitted acknowledgement packet and subsequent synchronization or finalization

packet, the ground station NERDP is now capable to create the appropriate size byte mask for the retransmission of packets.

After the transmission of every synchronization or finalization packet by the transmitter, NERDP utilizes timeout conditions specified by the user and the corresponding baud rate. This transitions the satellite to a listen state wherein it will only wait a limited window of time for a finalization packet or a missing packet request. The first timeout window is fully dependent on the number of packets sent in order to reduce the dead time the satellite waits for incoming data. If the timeout is reached, the synchronization or finalization packet is retransmitted, this time with the timeout reduced. This reduction in timeout happens because the other end of NERDP now no longer has to process a full frame of data, and only has to process a single packet. After two retransmissions, if the transmitter does not receive an appropriate packet, the connection is dropped and the transmission ends. The listening logic of every data packet has a long standardized timeout so as to maximize the data captured, and if the timeout is reached, then the connection terminates and the limited data is written to file. If the ground station NERDP does not receive an expected synchronization or finalization packet after sending a missing packet request, it will also send two retransmissions with short time outs. This also happens if no data is received immediately after sending a continuation packet. This behavior attempts to trigger transmission from the nanosatellite NERDP transmitter and prevent the connection from timing out from lack of receipt of data from the ground station.

These retransmission mechanisms are designed into NERDP to provide reliability in communication and mitigate errors in the data. Loss of data is not uncommon in poor connections, and the ability to retransmit the data and be able to reassemble the data in the correct order is crucial to the function of nanosatellites. NERDP reliability functionality is specifically designed to reduce the cost of data overhead and the need for constant change of state of the hardware radios from receive to transmit. By implanting bursts of packets as frames and a very light header, nanosatellites can throughput more data in their limited bandwidth and provide more functionality as research platforms.

3. Packet Integrity in NERDP

Packet integrity is crucial in noisy data as it allows the receiver to separate valid data from malformed data. Every packet in NERDP can be subjected to integrity checks to verify integrity, and these checks can deliberately drop malformed packets if data integrity is a high priority. Natively, NERDP integrates the use of a 16-bit CRC for integrity checks and validation. This CRC is directly built into the header and can be calculated either before or after the encryption of the data, depending on the implementation of the user.

Objects transmitted over NERDP can vary in the degree of integrity required to ensure their validity. While stronger forms of integrity checks other than a 16-bit CRC exist, NERDP depends on the size of the integrity check value being 16-bits. This value can be calculated by a 16-bit CRC or a checksum algorithm. The only caveat is that the same algorithm be used on both ends of the transmission.

Overall, data integrity in NERDP is integrated at the packet header level. The packet headers are designed with data integrity at the forefront and take up 50% of the total packet header. Design of NERDP is focused on noisy signals, and a small reliable integrity check is crucial to the functionality of NERDP.

Prioritizing the development of an integrity check value comes at a price in the functionality of NERDP. By increasing the steps necessary before transmission of the packets, NERDP increases the processing cost of transmitting data. Such a cost is unavoidable if integrity is to be implemented at the packet level. If an integrity check is integrated at the frame or object level the scheme risks requiring large number of retransmissions, and bandwidth usage should be prioritized over processing costs, especially if they are small integrity checks on each packet.

4. Encryption Integration

The advantage of having a modular design where the encryption scheme of the data is completely external to the actual functionality of NERDP is the ease of integration or separation of encryption from NERDP. This modularity gives NERDP the ability to

operate with different implementations of encryption, or utilize the OTP encryption module designed for it.

If a transmission utilizes OTP encryption, the acknowledgement packet provides an offset to the encryption and decryption module indicating where the OTP should be read from. At the end of every encrypted transmission, this offset is increased by the size of the object transmitted to select new data from the OTP.

Encryption and decryption can be carried out easily in the transfer of the data or can be taken out altogether if the designers choose to focus on other aspects of the protocol. This optional encryption helps NERDP become a standard in data transmission as it allows multiple listening stations to communicate with the nanosatellite without the need to exchange a OTP with all of them.

5. NERDP Information Assurance Posture

NERDP provides a solution to all three components of information assurance: confidentiality, integrity, and availability. Ideally a system must attempt to reach the highest possible standard for each one of these components, but unfortunately some implementations are costlier than others.

NERDP provides, with the integration of OTP encryption, a very high level of confidentiality at a low cost of processing but at a high cost in system hard disk space. The need for a large infrastructure to support his high level of confidentiality may be prohibitively expensive for some space craft designs. NERDP does provide the flexibility for the encryption of packets, frames, and objects depending on the implementation. By replacing the encryption module utilized when each packet is encrypted with the OTP with an alternate encryption mechanism, the system can remain confident. Ultimately the confidentiality of the NERDP system is only as strong as the encryption mechanism utilized.

Data integrity is provided on each packet by 16-bits in the packet header. NERDP does not require a specific integrity check, and any integrity check that provides a 16-bit checksum or signature of the data can be used. It is possible for NERDP to be modified to

use larger integrity checks. This will come at a cost of data overhead and is not recommended. Currently TCP and UDP headers do not require more than 16-bits to provide integrity of the data transmitted [10], [11]. NERDP design heavily mirrors the implementation of integration checks of both IP protocols.

Availability is the property of NERDP to provide the data to any authorized listener making a request. While there is currently no method to authenticate the users making the request from a nanosatellite employing NERDP, this can be mitigated with an authentication scheme that requires a password or authentication in the request package. This functionality can be implemented either within NERDP or at the application layer and requires future investigation. NERDP does provide the ability to mitigate data loss in the event of interference or error. This makes the data constantly available to any receiving platform.

Overall, the security posture of NERDP is strong given the limitations in nanosatellites processing and bandwidth. Finding a balance of all three components of information assurance is application and design-specific, so a quantifiable approach to evaluate the posture is difficult to assess. Nonetheless, NERDP provides the functionality typically associated with a larger protocol at a fraction of the data overhead. The ability to maintain the information assurance state on par with other protocols truly makes NERDP behavior an asset to nanosatellite communications.

D. PACKET HEADER STRUCTURE

The packet header for all NERDP packets, regardless of packet type is standardized to 4 bytes or 32 bits. These 32 bits contain all the information needed to route the packets to the correct port for processing, the integrity check value, and the information needed to reassemble the received packets in the correct order in every frame (Figure 20).

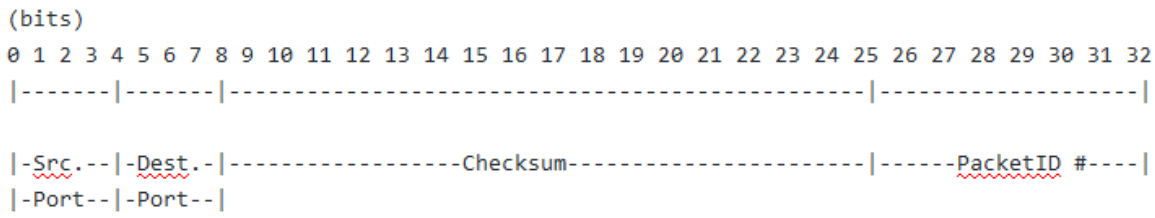


Figure 20. NERDP 32-bit packet header containing the source port, destination port, checksum, and the packet identification number

In order to properly identify the type of packet received, NERDP must route the packets to specific ports. These ports provide different logic and parsing for the data packets and some ports are specific for different functions. In the event that packets need to be sorted by their different source ports, NERDP must also be able to provide that functionality. NERDP packet headers are designed to include four bits of data for the source port and four additional bits for the destination port. Combined they form a total of one byte on the packet header and provide 16 destination and 16 source ports for communications.

Regardless of whether the integration is carried out pre- or post- encryption, or whether the data is even encrypted, two bytes are utilized for integrity check. The size of this checksum is drawn largely from the size of checksums in UDP and TCP [10], [11]. These 16 bits, or two bytes, provide the space for a cryptographic integrity check in the form of a checksum or CRC. NERDP does not enforce a particular type of integrity check, and like encryption, the calculations are done by a module that can be integrated or omitted depending on the implementation. By making the integrity check modular, NERDP can be treated independent of the integrity calculation. Regardless of the integrity check used, two bytes is half of the packet structure and should be more than enough to ensure integrity.

Finally, the last byte of the packet provides a packet identification number. This byte is used to dictate the order of the packet in the frame and is maintained by NERDP as the packet is transmitted. By limiting the packet number to one byte, NERDP is only capable of sending a maximum of 256 packets per frame. While this frame may be small compared to the thousands of packets some objects may require, due to the noisy signal

and high number of retransmissions expected, 256 packets is manageable frame size that ensures a more stable transfer of data. This is important if the connection is poor and the data requires a high level of integrity. If no integrity is required, the light exchange of two packets with no payload data (the synchronization and continue packets) to continue to the next frame are a small 8-byte price to pay relatively speaking to ensure reliability of the data (Figure 16).

Overall the design of the packet header is compact and provides the infrastructure for all of the key functionality of NERDP. Most of the packet header is dedicated to the integrity check due to its large size. The main goal behind the design of the header is to take the most common packet type, assumed to be the data type packet, and minimize the packet header. This minimization removes some functionality as a packet type, but this can be mitigated by the control packet design. By pushing some of the packet header that is not required for all packets and leaving only the essentials, the NERDP packet is a lightweight and feasible solution.

E. PACKET DESIGN

Pushing the some of the functionality into the data payload section of the packet is a design decision that leads to the creation of several packet types. These packet types are largely defined by their destination port and their data payload. Three classes of packets allow for the packet header to remain small when the packets do not require large overhead, and reduce the retransmission of unnecessary data within that packet. These decisions lead to a substantial reduction in the data overhead and still provide key functionality.

1. Control Packets

Control packets are used to communicate between nodes utilizing NERDP, and maintain the different states of the data transfer. These packets do not carry segments of the objects transferred between the nanosatellite and the ground station in their payloads. These packets communicate over port 0, and are used to request specific behavior or change the state of the transmission. Packets include the request, acknowledgement, synchronization, missing packet request, continuation, and finalization packets. The

packets are classified as control packets because their payload typically carries data relevant to the NERDP transmission of data, not the data itself. All have same four-byte packet header. The only control packet whose packet identification number in the NERDP header matters is the acknowledgement packet. This is due to the fact that it is the only control packet that may require retransmission at any point during the data transfer.

a. Request Packets (REQ)

Request packet payloads are sent from the receiver to the transmitter over port 0. These packet payloads are comprised of a three-byte string ‘REQ’ signifying the type of packet, a single byte requesting the port needed, and the name of the object requested by the receiver. The limitations on these values is that the port number must be a value 0–15 as only four bits are used for ports by the packet header, and the object name must be less than the total size of the packet minus the size of the ‘REQ’ type, and the byte used for the requested port.

b. Acknowledgement Packets (ACK)

Acknowledgement packets are used to acknowledge the packet request, and respond with metadata about the object that is to be transmitted to port 0 on the receiver and must have packet identification number of 0. This identification number is important because it is the only control class packet whose retransmission is treated like a data type packet. This packet payload consists of a three-byte ‘ACK’ string, followed immediately by an eight-byte unsigned long-long integer utilized to signal the OTP offset used for encryption, and are followed by an eight-byte unsigned long integer type that stores the size of the object requested. These values are made large enough to support large offsets and object sizes, but can always be modified if found too small as long as the parsing logic is modified accordingly.

c. Synchronization Packets (SYN)

Synchronization packets are used as a breakpoint to trigger the receiver to verify the packets received and begin retransmission if needed. These are sent to the receiver

over port 0, and aside from their three-byte ‘SYN’ string indicating the packet type, have a null payload. This is due to the fact that the ‘SYN’ message is all that is needed, and is a clear example of how packet overhead was reduced by eliminating that flag from the packet header and instead making it its own dedicated packet type under the control class.

d. Finalization Packets (FIN)

Finalization packets are used to signal the end of transmission over port 0, but have different effects depending on whether the transmitter or receiver transmit them. If a transmitter transmits a finalization packet, it is signaling to the receiver that the data transfer has reached the end of the object and is requesting if the receiver needs any retransmissions. If a receiver transmits a finalization packet regardless of the type received, it is signaling to the transmitter that the data transfer is finalized and to return to the initial listening state. Each finalization packet contains a three-byte ‘FIN’ string labeling the type of packet, and much like the synchronization packet it also contains an empty payload which helps reduce the total data overhead.

e. Missing Packet Request (MIS)

Missing packet request packets are sent from the receiver to the transmitter upon receipt of a synchronization or a finalization packet over port 0. These packets are used to request the burst retransmission of packets should the receiver need it. They have an identifier three-byte string ‘MIS’ followed by a 256-bit mask that signals which packets need to be retransmitted. This 256-bit mask makes the payload 32 bytes, regardless of the number of packets needed to be retransmitted or the size of the packets. This makes the ‘MIS’ type packet the largest packet in the control class. These 256 bits are used to represent the 256 packets sent in the immediately preceding frame. If the *nth* bit is a 1, this indicates that the *nth* bit is missing and should be retransmitted.

f. Continuation Packets (CON)

Continuation packets are used by the receiver to signal to the transmitter over port 0 that the retransmission is over, and to proceed with the next frame of packets. These packet types are only used if the retransmission was started with a synchronization packet

from the transmitter and not a finalization packet. The payload of the packet contains a three-byte string ‘CON’ and an empty null payload much like the synchronization and finalization packets

2. Data Packets (DAT)

Data packets are the most common packets. These packets do not require any kind of label as they are typically the only packets going to data ports 2 or higher. The payload is full to the maximum size of the object data, and the packet identification number is dictated by its sequential position as it is read from the object and transmitted. NERDP currently assumes all data sent to ports 2 or higher is a data packet.

3. State of Health Packets

State of health packets are designed to operate as one-way datagrams over port 1. These packets provide triangulation data and other metadata about satellite operations and do not undergo retransmission like control or data packets. These operate on their dedicated port so that ground station receivers can route and process the data differently.

a. State of Health Request (SRQ)

Operating over port 1 over an unreliable datagram, these packets are sent from the receiver to the transmitter to request state of health data. The payload consists of a three-byte identifier ‘SRQ’, and are otherwise empty.

b. State of Health Response (SRP)

Operating over port 1 over an unreliable datagram like the request packets, these packets are sent from the transmitter over the receiver as a response the state of health request packet. The payload consists of a three-byte identifier ‘SRP’ and are followed immediately by the state of health data.

F. NERDP PROOF OF CONCEPT PLATFORM DEVELOPMENT

In order to test the validity of the design of NERDP, a proof of concept was needed on a platform that could behave similarly to a nanosatellite. Using the same

virtual machine setup from the encryption module platform development, NERDP was implemented and deployed to evaluate its performance and demonstrate its feasibility as a protocol.

1. Mechanism Development and Platform

Utilizing two virtual machines identical to those in Chapter III connected over an IP version 4 (IPv4) network, NERDP was designed and implemented in Python 3.5.2. This configuration allowed quick development and testing of the NERDP proof of concept platform. This came at the added benefit of being user friendly and readable should other developers wish to implement NERDP in other languages and systems.

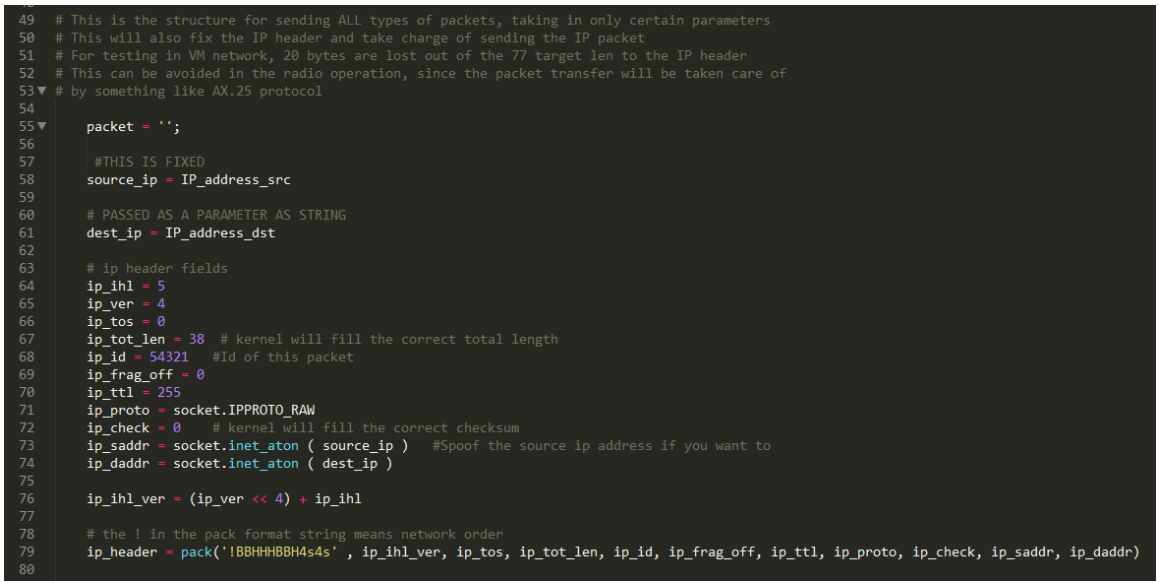
a. Packet Transfer in Platform

To simulate the behavior of NERDP in an environment absent of Layer 2 protocols like AX.25, the IPv4 layer is implemented as the packet transfer protocol. This implementation creates a raw socket that binds to a network interface and sends raw data over the network (Figure 21).

```
31  # create a raw socket that will bind to the network interface, this will receive all raw packets at the OSI layer 3
32
33  s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_RAW)
```

Figure 21. Implementing a raw socket for the transfer of raw bytes over a network in Python 3.5.2

This socket does not automatically populate the IPv4 header needed to transfer the packets through the network to the machine with the corresponding destination IP address. To enable this, the IPv4 packet header is implemented manually (Figure 22).



```
# This is the structure for sending ALL types of packets, taking in only certain parameters
# This will also fix the IP header and take charge of sending the IP packet
# For testing in VM network, 20 bytes are lost out of the 77 target len to the IP header
# This can be avoided in the radio operation, since the packet transfer will be taken care of
# by something like AX.25 protocol

    packet = '';

     #THIS IS FIXED
    source_ip = IP_address_src

    # PASSED AS A PARAMETER AS STRING
    dest_ip = IP_address_dst

    # ip header fields
    ip_ihl = 5
    ip_ver = 4
    ip_tos = 0
    ip_tot_len = 38  # kernel will fill the correct total length
    ip_id = 54321   #Id of this packet
    ip_frag_off = 0
    ip_ttl = 255
    ip_proto = socket.IPPROTO_RAW
    ip_check = 0    # kernel will fill the correct checksum
    ip_saddr = socket.inet_aton ( source_ip )   #Spoof the source ip address if you want to
    ip_daddr = socket.inet_aton ( dest_ip )

    ip_ihl_ver = (ip_ver << 4) + ip_ihl

    # the ! in the pack format string means network order
    ip_header = pack('!BBHHHBBH4s4s' , ip_ihl_ver, ip_tos, ip_tot_len, ip_id, ip_frag_off, ip_ttl, ip_proto, ip_check, ip_saddr, ip_daddr)

```

Figure 22. IPv4 packet header for use with a raw socket, all options and values have to be implemented manually and packed into the correct structure

This packet header contained all of the needed infrastructure to simulate the packet being sent from one machine to another. On the receiving end, a listener receives a tuple with the IP address source and the data in raw byte format.

b. Sending Packets

A function titled *sendPacket* was created to send the packets in a generic format to facilitate the development of the different types of packet classes and types supported by NERDP. This function ingests the destination IP address, the source IP address, the packet identification number, the packet type, the payload, and the requested port and automatically creates the IPv4 header described above. The function takes the packet type and populates the payload and port values accordingly and sends the packet over the IPv4 network to the recipient. Appendices B and C have the implementation in detail. The similarities in packet structure are evident under the *sendPacket* function allow the different classes of packets to be put under the same sending function. The *sendPacket* function allows for each packet sent to contain the same components and facilitates the modularity of the code (Figures 23–26).

```
42  def sendPacket(IP_address_dst,IP_address_src, packetID, packetType, payload, reqPort):
43
44  # PAYLOAD MUST BE OF TYPE BYTES SO IT CAN BE ENCODED LATER
45  # IP_address_* MUST BE OF TYPE STR
46  # packetID MUST BE OF TYPE INT
47  # packetType MUST BE OF TYPE STR
```

Figure 23. Packet sender function input arguments

```
97      # Packet type used to request object
98      if packetType == 'REQ':
99          # Request must include a data port (not 0,1,2) on which the data will be sent
100         reqPortByte = (str(reqPort).encode('ascii'))
101         reqPortByte = pack('B',reqPort)
102         data = packetType.encode('ascii') + reqPortByte + payload # REQ (3 bytes), payload = REQUESTED PORT+objectname
103         # REQ packets get sent from port 0 to port0
104         srcport = 0
105         dstport = 0
106         portByte = (srcport<<4)+dstport
107
108         checksum = 0 # TODO: write a function to calculate checksum of payload
109         # Pack the header to 4 bytes total
110         NERDP_header = pack('!BHB', portByte, checksum, packetID)
111         s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));
112
113     # Packet type used to acknowlege request packet
114     if packetType == 'ACK':
115         data = packetType.encode('ascii') + payload #payload = ACK, OTP_OFFSET (8 bytes) , OBJ_SIZE (8 bytes)
116         # ACK packets get sent from port 0 to port 0
117         srcport = 0
118         dstport = 0
119         portByte = (srcport<<4)+dstport
120
121         checksum = 0 # TODO: write a function to calculate checksum of payload
122         NERDP_header = pack('!BHB', portByte, checksum, packetID)
123         s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));
124
125     # Packet type used to signal end of transmission, whether from EOF or if the ground station terminates
126     if packetType == 'FIN':
127         data = packetType.encode('ascii') + payload #empty payload
128         # FIN packets get sent from port 0 to port 0
129         srcport = 0
130         dstport = 0
131         portByte = (srcport<<4)+dstport
132
133         checksum = 0 # TODO: write a function to calculate checksum of payload
134         # Pack the header to 4 bytes total
135         NERDP_header = pack('!BHB', portByte, checksum, packetID)
136         s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));
137
138     # Packet type used to synchronize and request any retransmissions of the 255 packet frame
139     if packetType == 'SYN':
140         data = packetType.encode('ascii') + payload #payload = SYN, NULL
141         # SYN packets get sent from port 0 to port 0
142         srcport = 0
143         dstport = 0
144         portByte = (srcport<<4)+dstport
145
146         checksum = 0 # TODO: write a function to calculate checksum of payload
147         NERDP_header = pack('!BHB', portByte, checksum, packetID)
148         s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));
149
150     # Packet type used to indicate the missing packets and request retransmission
151     if packetType == 'MIS':
152         data = packetType.encode('ascii') + payload #payload = MIS, Packet numbers where each byte is one packetID
153         # MIS packets get sent from port 0 to port 0
154         srcport = 0
155         dstport = 0
156         portByte = (srcport<<4)+dstport
157
158         checksum = 0 # TODO: write a function to calculate checksum of payload
159         # Pack the header to 4 bytes total
160         NERDP_header = pack('!BHB', portByte, checksum, packetID)
161         s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));
162
163     # Packet type used to initiate transmission of next 255 packet frame and continue data transmission (end of retransmission)
164     if packetType == 'CON':
165         data = packetType.encode('ascii') + payload #payload = MIS, Packet numbers where each byte is one packetID
166         # CON packets get sent from port 0 to port 0
167         srcport = 0
168         dstport = 0
169         portByte = (srcport<<4)+dstport
170
171         checksum = 0 # TODO: write a function to calculate checksum of payload
172         # Pack the header to 4 bytes total
173         NERDP_header = pack('!BHB', portByte, checksum, packetID)
174         s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));
```

Figure 24. Control class packet structure implementation in Python 3.5.2 within the packet sender function

```
176     # Packet type used to indicate payload data packet
177     if packetType == 'DAT':
178         data = payload # payload is the data being sent
179         # packets get sent from port 2 to requested port
180         srcport = 2
181         dstport = reqPort
182         portByte = (srcport<<4)+dstport
183 
184         checksum = 0 # TODO: write a function to calculate checksum of payload
185         # Pack the header to 4 bytes total
186         NERDP_header = pack('!BHB', portByte, checksum, packetID)
187         s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));
188 
```

Figure 25. Data type packet structure implementation in Python 3.5.2 within the packet sender function

```
189     if packetType == 'SRQ': #request
190         data = payload.encode('ascii') # payload = 'SOHREQ'
191         # SOH packets get sent from port 1 to port 1
192         srcport = 1
193         dstport = 1
194         portByte = (srcport<<4)+dstport
195 
196         checksum = 0 # TODO: write a function to calculate checksum of payload
197         # Pack the header to 4 bytes total
198         NERDP_header = pack('!BHB', portByte, checksum, packetID)
199         s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));
200 
201     if packetType == 'SRP': #response
202         data = packetType.encode('ascii') + payload # payload = 'SOHRSP' + data of SOH
203         # SOH packets get sent from port 1 to port 1
204         srcport = 1
205         dstport = 1
206         portByte = (srcport<<4)+dstport
207 
208         checksum = 0 # TODO: write a function to calculate checksum of payload
209         # Pack the header to 4 bytes total
210         NERDP_header = pack('!BHB', portByte, checksum, packetID)
211         s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));
```

Figure 26. State of health type packet structure implementation in Python 3.5.2 within the packet sender function

In order for Python 3.5 to deliver the raw bytes correctly, their encoding had to be changed. This was done by using the pack function and specifying the various sizes and types of the data being stored. This way, all packets were treated as containing raw binary data and were sent over the raw socket.

c. Receiving and Parsing Packets

Receiving and parsing packets require knowledge of the payload received. Receiving the raw packets is left to the IPv4 socket, since it is assumed that an AX.25 implementation delivers the raw data to the NERDP layer. The packet is a tuple of the IP source IP address and the entirety of the packet. This packet contains the IPv4 header, the NERDP header, and the NERDP payload. Parsing the packets requires knowing the offsets of the data and storing it in the appropriate variables for later usage (Figure 27).

```
260             # PAYLOAD LENGTH HERE IS SET AT 77 FOR THE MODEL USED
261             # CAN BE CHANGED TO N BYTES.
262             # recieve the readLen number of bytes
263             packetRcvd = s.recvfrom(readLen)
264             # get rid of the IP header
265             packetRcvd = packetRcvd[0]
266             # unpack the portbyte and get the src and dst ports
267             portByte   = format(int(packetRcvd[20]),'02x')
268             srcport = int(portByte[0],16)
269             dstport = int(portByte[1],16)
270             # get the checksum (2 bytes)
271             checksum   = packetRcvd[21:23]
272             # packetID # (between 0-255)
273             packetID   = packetRcvd[23]
274             # the rest is payload
275             payload    = packetRcvd[24:]
```

Figure 27. Generic parsing for all packet data received

Packet specific parsing was then done depending on the destination port, packet type, and other class dependent structures. The full implementation for the ground station receiver can be found in Appendix B, and the implementation for the nanosatellite transmitter can be found in Appendix C.

2. Protocol Operation in Test Platform

Operation of the proof of concept protocol implementation was tested under various conditions to verify its functionality. While NERDP is designed with a various functionality, the test platform largely focused on the repeatability of the data and treated the integrity check and encryption scheme as independent modules. This focus allowed the evaluation of the base properties of NERDP.

The test platform was developed to transfer files of arbitrary size and content from one virtual machine to another. This operation consisted of a transmitter operating on one virtual machine as the nanosatellite, completely automated, while the ground station was modeled as if a user were operating the console through a keyboard. The user specifies the object to be transferred and the requested port. NERDP on the transmitter side automatically sends the request packets and receives the data and carry out its functionality. Once the receiver returns to its initial state, the user verifies the data received to ensure successful transfer. Additionally, using the Wireshark network traffic monitoring software, the user could capture and inspect independent packets to verify and debug packet behavior. Under normal operations, the current version 0.1.1 of NERDP proof of concept platform supports the transfer of files of arbitrary size, including empty files. Results of transfer are discussed in Chapter V, Results and Analysis.

In order to test the repeatability functionality of NERDP, several unit tests were written into the implementation of the test platform. These tests purposefully drop specific packets in the transmission and record the behavior of NERDP. These packet losses could be due to failure to meet integrity checks, or simply packet loss in the noisy signals. Drops were triggered for the acknowledgement packet and the data packets, and both packets at once. NERDP test platform version 0.1.1 currently does not support the use of state of health packets, nor does it support timeouts and retransmissions of synchronization, continuation, and finalization packets. That functionality is forthcoming in version 0.2.0. Results of the testing operation of the retransmission capabilities of NERDP test platform version 0.1.1 are discussed in Chapter V, Results and Analysis.

G. EVALUATING PERFORMANCE OF NERDP

Evaluating the NERDP functionality required the establishment of certain performance metrics for the protocol and the test bed. The test bed version 0.1.1 focused on the base operation and retransmission of specific types of packets. In order to evaluate its performance, its data overhead as a function of total data transmitted, discounting the IPv4 header) was assessed. Furthermore, the behavior of the retransmissions, the

preservation of test platform state, and its impact on data overhead compared to other protocols was also assessed

1. Data Overhead Compared to TCP, UDP, CSP

Performance comparison methods were calculated for NERDP, TCP, UDP, and CSP. These metrics listed the functionality, the data overhead costs for base operation, and the data overhead costs per packet retransmission. These metrics were established to evaluate the feasibility of using NERDP as a viable replacement for any of those protocols under limited bandwidth conditions. Some factors excluded from the comparison methods were the processing time and delivery latency per packet. These metrics were considered to be too hardware and datalink dependent, and should be investigated in a more realistic test bed with actual nanosatellites. Results of the evaluation and comparison are found in Chapter V, Results and Analysis.

2. Reliability as a Data Loss Mitigation Method

The utilization of reliability as a mitigation for data loss is a common occurrence. NERDP's ability to mitigate data loss was tested at the packet and frame level. It should be noted that in order to obtain more comprehensive data, deletion of data at the byte and bit level should also be carried out. While no difference is expected of the behavior of NERDP in data loss, as discussed in Chapter V, Results and Analysis, NERDP behavior must be modeled extensively to ensure the performance meets the design criteria and expected behavior.

H. MAKING NERDP OPEN SOURCE

Nanosatellites and small satellites such as CubeSat have a long history of being developed using COTS and open source components [15]. Development of the NERDP test platform and the protocol design follow in the footsteps of the development of other nanosatellite and small satellite missions. To this end, NERDP is designed with the community in mind, and is meant to be shared openly as a standard and solution for nanosatellite developers. The proof of concept implementation in Python 3.5.2 using a Linux platform allows for a broad distribution and collaboration between researchers and

development. This design choice allows NERDP to be versatile enough for any mission, while still maintaining clear key components designed at the Naval Postgraduate School.

I. CHAPTER SUMMARY

Based on the environment and conditions within which nanosatellites operate, NERDP is proposed as a satellite communication scheme that provides much needed functionality at a fraction of the data overhead and complexity as other protocols. This design is oriented with bandwidth efficiency as its driving force. The design provides data reliability, integrity, and support for data confidentiality in a small lightweight packet. Furthermore, a proof of concept test platform for developers is created and provided to facilitate the development and integration of the protocol in future nanosatellite missions. This test platform includes several mechanisms to verify and test its functionality and is provided as a free open source tool. While NERDP does have some limitations such as the limited number of ports available, and the need for a Layer 2 transport mechanism, the design is robust, flexible, and simple enough that future nanosatellite and small satellite missions that choose to implement it will benefit greatly from it.

V. RESULTS AND ANALYSIS

A. INTRODUCTION

Before NERDP can be implemented on nanosatellites it must be put through rigorous testing and analysis to ensure behavior is predictable and desirable. Utilizing the platforms for a proof of concept, an evaluation of the system performance can be made. This evaluation isolates the specific performance metrics of interest while laying a foundation for increased functionality development and experimentation of the NERDP design.

Measuring the performance of NERDP is closely tied to the design goals of the protocol and the role it seeks to play in nanosatellite communications. The limitations of the environment dictate the goals of the protocol design and are reflected in the choice of performance metrics. An evaluation of these metrics helps determine the applicability of the protocol suite.

B. OTP ENCRYPTION MECHANISM EVALUATION

Evaluating the OTP encryption mechanism relies heavily on measuring the mechanisms performance as defined by the size of the encrypted files before and after encryption, robustness to error propagation, and complexity and processing costs. Establishing these metrics and evaluating their generic behavior on a testbed allows researchers and developers to determine the feasibility of utilizing the encryption mechanism in their satellite development and their own integrated system.

1. Size of Cipher Text and Plain Data Text Analysis

As described in Chapter III, some encryption mechanisms utilize specific block sizes for their encryption. This results in larger cipher text sizes in comparison to the plain text data sizes. Mechanisms like AES-128 have block sizes of 128 bits or 16 bytes [19]. This addition of 16 bytes in packet radios that utilize less than 100 bytes results in additional packets of data being transmitted which impact the window of availability time frame. While the impact may not be very substantial, it is still an inefficiency that can be

mitigated, and in the event of sending multiple small packets of data that need to be encrypted, result in an accrued increase of unnecessary data.

Utilizing the testbed used to develop the OTP encryption mechanism, data sizes before and after encryption are compared. OTP encryption encrypts one byte at a time instead of blocks of multiple bytes, so calculating and verifying in the testbed results in a predictable file size. Block cipher encryption results in file sizes that are multiples of the block sized used. This leads to the conclusion that as long as the data is not an even multiple of the block size of another cipher, OTP will still be the most efficient data size after encryption. The difference between the sizes is clear when the size of the cipher text is plotted as a function of plain text size. Block ciphers result in a stepwise function whose interval is a multiple of the block size used to encapsulate the whole message, while OTP encryption results in a linear one-to-one function (Figure 28).

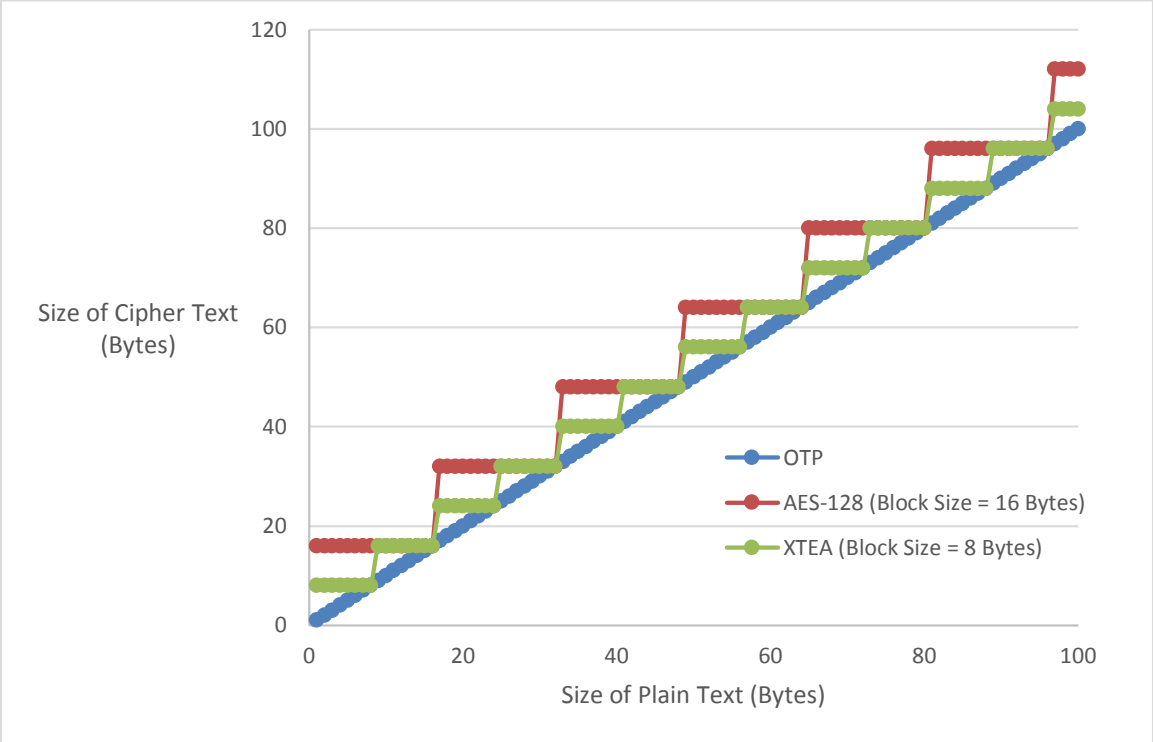


Figure 28. Comparison of cipher text sizes as a function of plain text size for 3 different cipher block sizes and OTP

The results are conclusive and constant regardless of the size of the plain text. The addition of data, while it may seem small, should be taken into consideration when selecting the encryption mechanism. Of the three mechanisms recorded and tested, OTP was the smallest and most efficient when it came to the size of the cipher text as a function of plain text size. This is due to the fact that OTP's block size is effectively one byte as it encrypts each byte independently.

2. Error Insertion Results and Analysis

An evaluation of the OTP encryption mechanism's resilience to error insertion was carried out following the methodology laid out in Chapter 3. Looking at the three methods of data manipulation insertion, deletion, and replacement, it was found that data deletion and insertion led to catastrophic results for OTP encryption. The errors were again repeated in block ciphers to similar conclusions.

In the event of data deletion or insertion, all data after the error location was unreadable and ineffective. AES-128 and XTEA operating in counter mode limited the propagation of the error to the location of the error and all subsequent data and did not affect the preceding data even if the data was in the same block. OTP encryption had similar results as each byte is encrypted independently and any insertion of data affected all subsequent bytes. These results did not favor one encryption mechanism over the other as they were all affected equally so researchers and developers should focus on other metrics for encryption.

In the event of replacement of data, the error propagation from all three methods of encryption in single bit flipping was a 1-to-1 ratio. For every byte affected in the plain text, there was equivalently 1 byte affected. When expanded to flipping bits in bursts, as expected if the burst happened in the "fault line" between two bytes, both bytes were affected in the plain text. Again these results did not favor one mechanism over another.

Overall, looking at both block ciphers and OTP encryption error resilience did not provide a valuable metric to favor one encryption mechanism to another. Each of the three mechanisms tested for error resilience had the same pitfalls and strengths. All three mechanisms were vulnerable to data insertion or deletion as they created a "shift" in the

data in the case of data replacements, all mechanisms managed to limit the data replacement from propagating to the byte where the data occurred. The OTP encryption mechanism testbed with random bit flips can be found in Appendix A.

3. Processing Costs and System Complexity

Evaluating processing costs relies heavily on the number of iteration rounds undertaken by the encryption mechanism. Each iteration costs processing time, and the more complex each iteration is, the more the cost escalates. While OTP only needs one iteration per byte, AES-128 requires 10 iterations per block, and XTEA requires 64 iterations per block [14], [15]. Assuming initially that all rounds are equally expensive to encrypt the same data, calculating the number of iterations as a function of plain text size reveals that as expected XTEA is the most expensive to compute. AES-128 and OTP on the other hand begin as expected and start to diverge as OTP increases as a linear function, and AES-128 grows as a step-wise function. It seems that focusing on iterations alone, AES-128 may be the better cipher mechanism (Figure 29).

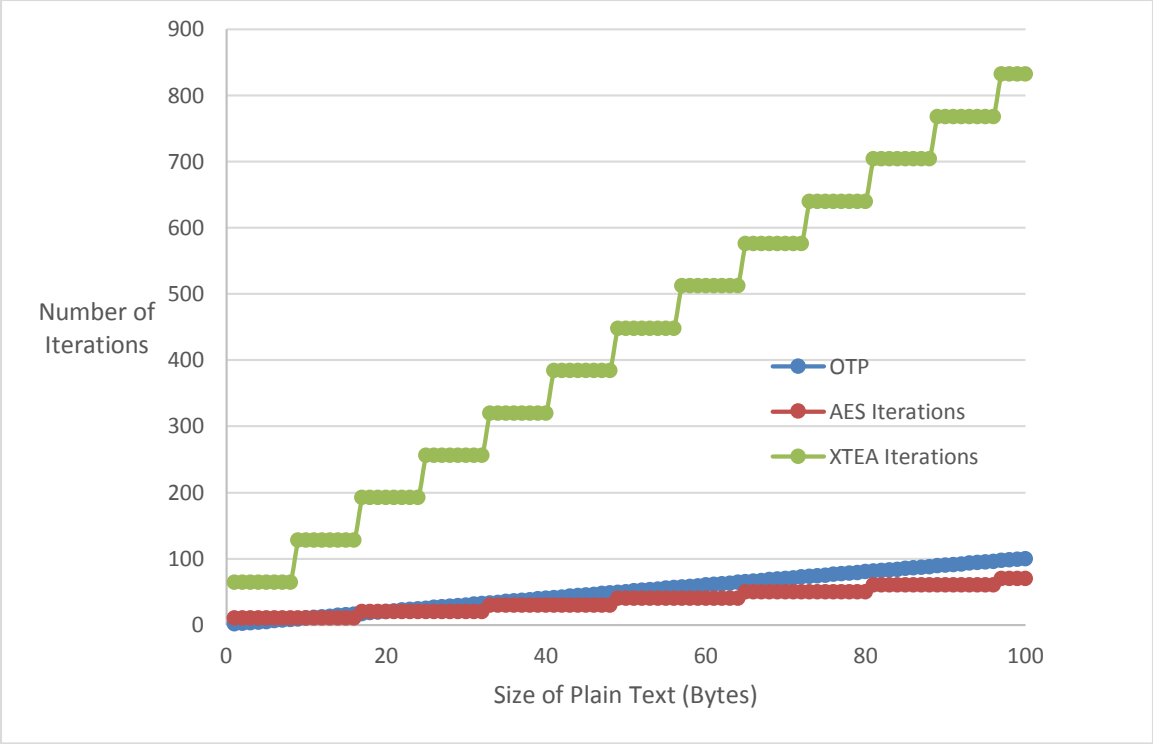


Figure 29. Number of iterations for OTP, XTEA, and AES-128 as a function of plain text size

These results are deceiving as they do not accurately reflect the processing cost of each iteration. Looking specifically at the difference between iterations in AES-128 and OTP, the number of operations within each iteration is different. Excluding reading and writing from a buffer for both mechanisms and focusing only on the actual operations undertaken by the cipher, OTP only utilizes one operation: the logical XOR. AES-128 utilizes four different functions each with their own multiple steps on each iteration [24]. Assuming that each function takes as long as the logical XOR, the performance of AES-128 changes dramatically. The number of functions undertaken by AES greatly increase the cost of each iteration and thus become more expensive than OTP encryption (Figure 30).

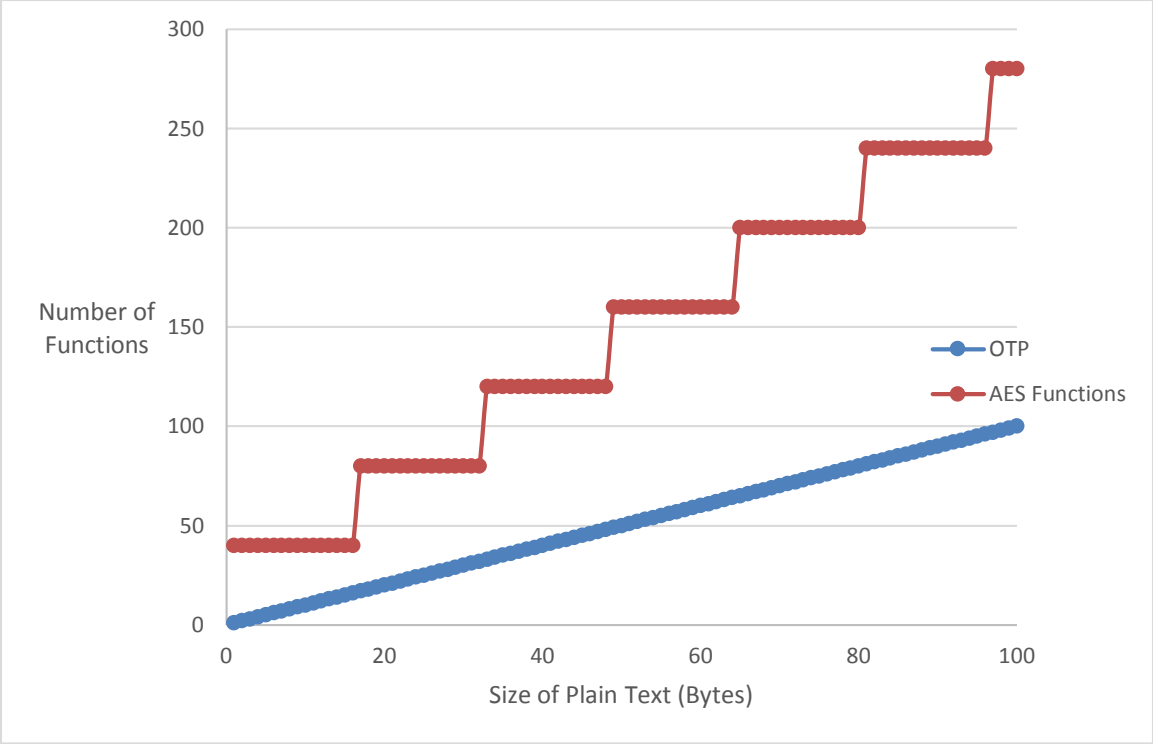


Figure 30. Number of functions taken by AES-128 and OTP as a function of plain text size

While these results could be extrapolated to ensure their conclusiveness, these results favor OTP encryption over AES-128 due to the lower number of functions required. Generating and analyzing these results on the specific hardware to be used in the nanosatellite would provide a much better data set and performance metric.

Finally, evaluating the system complexity and key infrastructure is important in nanosatellites and small satellites. This is due to the size, weight, and volume limitations in addition to the limited processing and memory availability to the computational package. In this key aspect, despite the fact that OTP encryption is lightweight in its implementation, OTP encryption requires a large amount of disk space equivalent to the total data sent by the nanosatellite in its lifetime. These factors are exacerbated by increased data rates and prolonged mission lifetime. On the other hand, symmetric key encryption mechanisms with 128-bit key spaces have a much smaller infrastructure cost. While these costs can be mitigated, in this metric OTP encryption is the less ideal choice.

C. NERDP SYSTEM EVALUATION

Evaluating the performance of NERDP is reliant on the performance under two circumstances: base operation, and data loss operation. In base operation NERDP is evaluated for the data overhead measurements in comparison to TCP, CSP, and UDP. Due to the limited documentation on CSP and its limited implementation, calculations for its overhead were treated similarly to TCP, especially for retransmission and acknowledgement. Further investigation is required into CSP functionality to ensure that the models are correct. Note that TCP and UDP use 20 bytes of data overhead for all of the transmissions. While the analysis assumes that all protocols are fit into a 77-byte packet encapsulated with AX.25, of those 77 bytes 20 must be used for the IP header if the packet is running on TCP or UDP. These protocols are reliant on the IP header for functionality and without this header data cannot be routed.

1. Data Overhead Metrics under Base Operation

Looking at base operation with ideal circumstances and no data loss, measurements were calculated and verified with the testbed for NERDP, and calculated for TCP, UDP, and CSP for files of 100 bytes, 1000 bytes (kilobyte), 1000000 bytes (megabyte), and 1000000000 bytes (gigabyte). TCP and CSP calculations accounted for the three-way handshake to establish a connection, and the four packet connection close. Similarly, NERDP calculations include the request and acknowledgement packets, and the finalization sequence packets. All other control packets within the data stream are accounted for and the overall data transfer is measured regardless of direction. All packets and their overhead are recorded. The data for base behavior at low data sizes seems to favor UDP, CSP, NERDP, and TCP in descending order. This phenomenon is surprising, yet expected due to the fact that NERDP has a higher initial cost than CSP when establishing the acknowledgement packet that within itself is 19 Bytes. As the file sizes grow, the data behaves as expected with NERDP accruing lower data costs than even UDP. In comparison to TCP, NERDP has over one order of magnitude smaller data overhead. The performance of NERDP as an alternative to TCP is clear especially as data sizes increase (Figure 31).

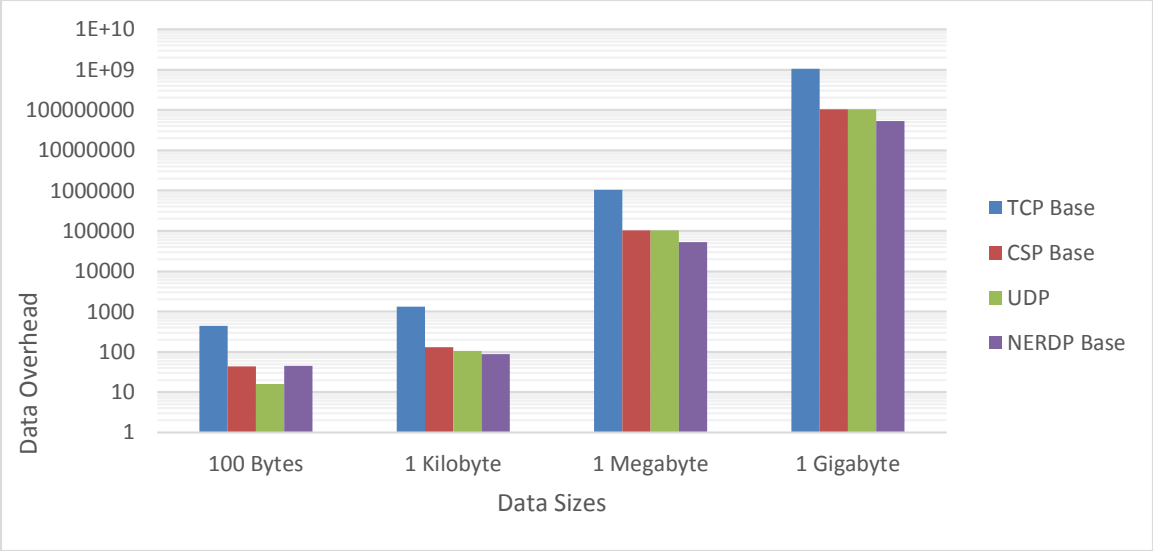


Figure 31. Data overhead as a function of object data size for base behavior of TCP, CSP, UDP, and NERDP

These results validate the hypothesis that NERDP can serve as an alternative implementation of TCP. NERDP data overhead is significantly less than the TCP implementation, while providing the same functionality. Disregarding the CSP results, reveals the verified comparison of TCP, UDP, and NERDP and the clear advantage NERDP has over TCP (Figure 32).

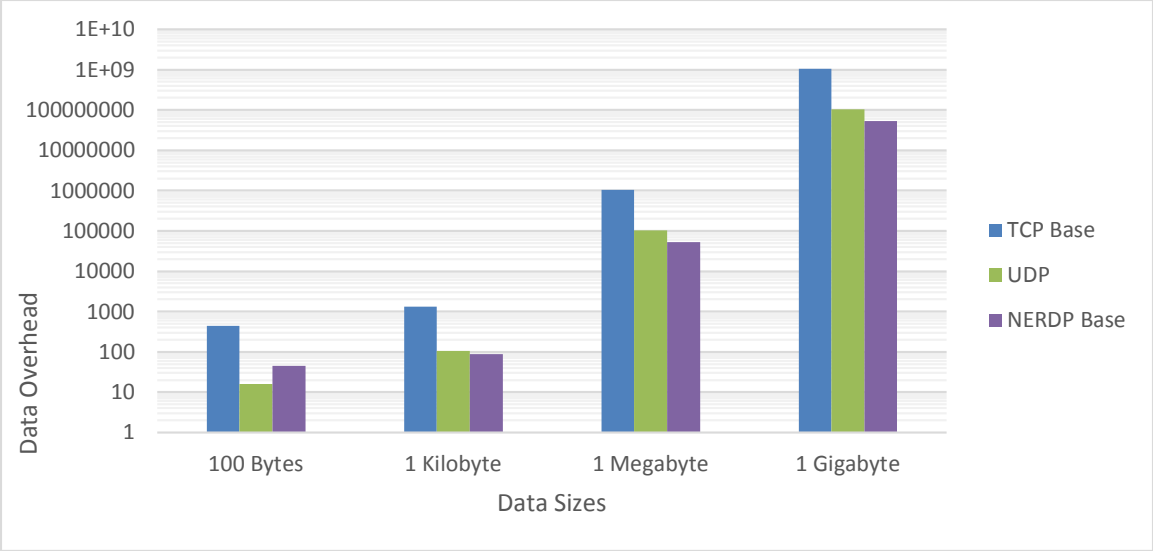


Figure 32. Data overhead costs as a function of various data sizes for base behavior of TCP, UDP, and NERDP

2. Data Overhead Metrics under Data Loss Operation

Continuing the same assumptions as for base operation, data loss operation metrics represent a "cost of reliability" per packet. This cost represents on average the cost in data overhead to retransmit a packet in TCP, CSP, and NERDP. UDP is excluded from this performance metric due to the lack of reliability functionality in UDP.

The cost of reliability is calculated by adding the data overhead needed to retransmit 1, 10, 100, 1000, and 10000 packets. In the case of NERDP, performance varies as burst retransmission means all packets may be in the same frame, or may be in multiple frames. In order to get a clear understanding of the variation, the minimum and maximum amount of frames for NERDP are represented. For TCP and CSP calculations, the cost of reliability data overhead is equal to twice the data overhead for a packet multiplied by the number of packets retransmitted. NERDP calculations require frame calculations. To calculate the least amount of overhead, the number of packets are put in the least number of frames as possible to minimize the synchronization and missing packet requests. To maximize the number of data overhead, each packet is assumed to be in its own frame exacerbating the problem. The results are evident that NERDP varies greatly depending on how many packets are missing per frame (Figure 33). Eliminating

CSP data sets shows that NERDP can be costlier than TCP under the right circumstances (Figure 34).

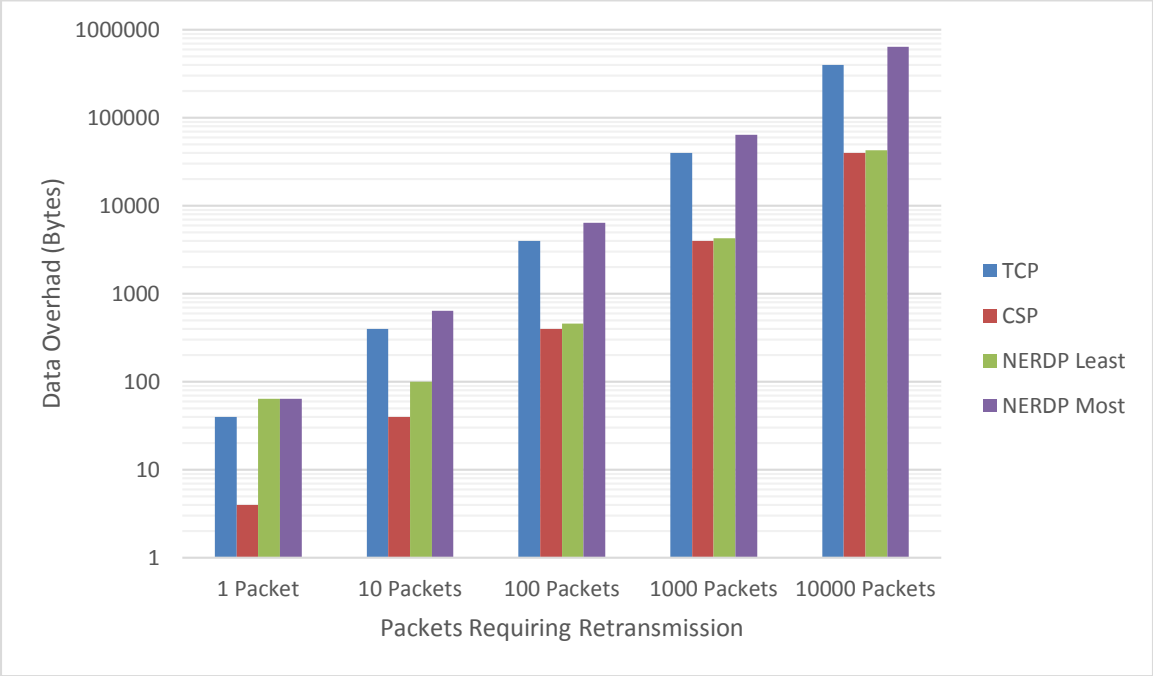


Figure 33. Data overhead cost of retransmission per packet for TCP, CSP, and the least and most possible values for NERDP

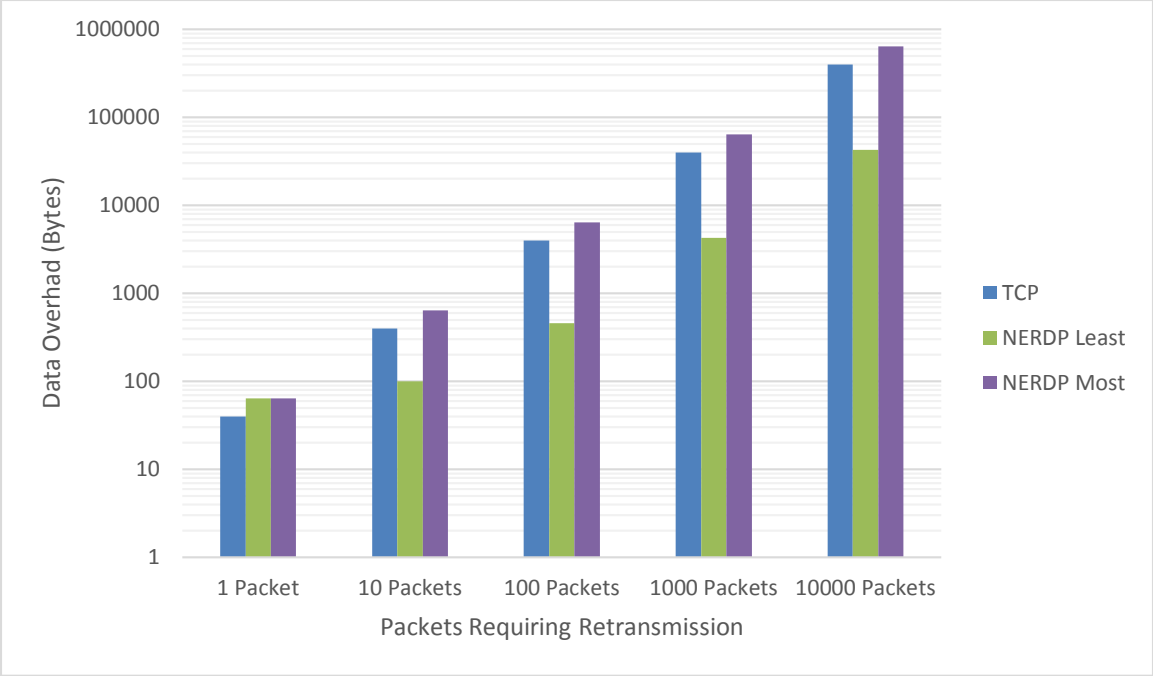


Figure 34. Data overhead costs for retransmission per packet for TCP and the least and most possible values of NERDP

Throughout all methods of operation, the analysis of the data overhead indicates that NERDP is a feasible alternative to other protocols. The maximum accrued data overhead accumulates only under very poor transmission situations which seem unlikely as burst errors are common. The platform developed supports the hypothesis that the design of NERDP is sound and provides functionality at a fraction of the TCP cost. This hypothesis comes with the caveat that TCP is still a better implementation if the number of packets lost in a transmission is close to one, and if the remainder of the retransmissions are spread out over multiple frames in NERDP. Further analysis should be done to evaluate the breakeven point where the two protocols accrue the same cost.

D. DRAWBACKS

Because NERDP operates in frames, with at least 2 packets being needed in between frames, overtime, NERDP has a smaller throughput of data than TCP. This means that looking strictly at the number of packets transmitted by both protocols, for every 256 packets, NERDP loses 2 packets of data to control packet exchange. This

drawback is put in perspective by considering that NERDP utilizes less data overhead, and as such is able to increase the payload size of the packets and reduce the number of packets transmitted. This increase of payload data is crucial especially in these limited packet sizes, so while TCP is faster, NERDP requires less data packets for transfer. Ideally, the timing analysis of NERDP should be performed in a realistic environment and platform, and the results compared to TCP. The Python platform developed did not allow for this type of analysis, as Python operates at a slower rate than other lower level languages.

E. FINANCIAL AND PROCESSING SYSTEM COSTS

By making a software solution for communications to replace already existing protocols and making it an open source COTS component, the financial cost to implement NERDP is small. The implementation may impact the actual system processing costs depending on the processing capability of the satellite and of the efficiency of implementation. Protocols like IP and TCP can typically be handled by the kernel without the need for much interaction from either the satellite or the ground station operator. In order to maximize efficiency, low level programming languages like C or x86 Assembly are recommended which will make the implementation and processing costs much lower than a high level language such as Python. The performance of the implementation of NERDP is left to the designer of the communications subsystem.

F. OVERALL SYSTEM INFORMATION ASSURANCE POSTURE

Originally NERDP started out as a mechanism with which to introduce confidentiality and information assurance into insecure channels typically used by nanosatellites and ground stations. As research proceeded, the integration of functionality supported availability and integrity checks at the packet level which helped provide a vulnerability assessment of the entire security posture of current nanosatellite systems.

1. Confidentiality Vulnerability Assessment

Currently, there is no clear standard for confidentiality in nanosatellite communications, and the closest existing standard is the usage of XTEA in CSP. The

overall lack of standardization and implementation of an encryption scheme is limited by the high data error rate in the data stream.

The lack of a standard encryption scheme for nanosatellites is of concern, as it impacts the sensitivity of the experiments. Furthermore, it allows third party observers and adversaries to intercept and record the data being transmitted. Allowing unauthorized users to listen and record the data from the broadcast without needing any authentication results in other parties obtaining the data at no cost. This limits the ability of institutions to carry out proprietary research as they are vulnerable to interception. The lack of encryption also leaves ground stations vulnerable to man-in-the-middle attacks where an adversary can intercept or spoof the data and make the ground station believe it is connected to the nanosatellite, while feeding malicious packets to the ground station. These malicious packets are indistinguishable from normal packets as they will pass integrity verification.

Encryption mechanisms help make attacks like man-in-the-middle difficult to succeed. The costs of these are the biggest deterrent to the implementation of confidentiality mechanisms in nanosatellites. In order for encryption mechanisms to work well, they require reliability, and to implement reliability, IP/TCP is the current standard. This standard comes at a high bandwidth expense which forces nanosatellite designers to sacrifice confidentiality in favor of data throughput. NERDP provides infrastructure for OTP encryption and also provides reliability at lower costs than IP/TCP. Research, design, and evaluation of performance indicates that NERDP could facilitate the integration of confidentiality mechanisms into nanosatellite communications. This significantly increases the confidentiality posture of nanosatellite communications. By integrating OTP encryption into the NERDP infrastructure, the protocol provides a valuable strength by limiting the dissemination of the encryption key to just the ground station and the nanosatellite. By using truly random data, and limiting its availability to two locations, one of them in LEO, the availability for adversaries to circumvent the encryption is severely constrained.

2. Integrity and Availability Vulnerability Assessment

All existing protocols provide some measure of integrity to verify the contents of the packets. In the case of TCP and UDP, integrity is typically handled by the kernel and does not reach the user or nanosatellite. If a packet is malformed it is immediately dropped, and in the case of TCP immediately requested for retransmission. This functionality is very connection based and assumes that if a packet is malformed it can always be retransmitted easily. In the case of nanosatellites, if the connection is very degraded, it may be wise to not discard all packets that fail integrity checks. Being able to specify whether or not a protocol should ignore or enforce integrity rules could be very beneficial in the data collected. NERDP offers the functionality to discard integrity checks and keep all packets sent, while still retaining the capability to discard packets whose integrity fails. The standards for integrity are just as robust as TCP and UDP, but more flexible. If a packet is retransmitted multiple times, NERDP can store all multiples of that packet for further review and continue the transmission. This is helpful if the data corruption is happening at a level other than transmission. While adversaries may purposefully alter the data inflight triggering a loop of retransmissions from the NERDP transmitter, limiting retransmissions can be used to combat a potential infinite loop. Ultimately, the usefulness of integrity checks is application dependent as opposed to strict implementations defined by a kernel that believes it is in a connection based network on the ground. While NERDP is neutral relative to the strength of data integrity and verification, it provides designers and users more flexibility and finer control on its implementation.

Availability in nanosatellites is complicated due to the large number of external factors that impact the availability of data connections between ground stations and nanosatellites. Due to small on orbit transmitter power, signals can be weak, noisy, and unreliable. Current protocol implementation focuses on improving availability of Layers 1 and 2, while disregarding higher protocols. Nanosatellite signals in UHF and VHF are weak, unreliable, and vulnerable. An adversary can physically jam or drown the signal and prevent ground stations from communicating with the nanosatellite. Furthermore, an adversary could oversaturate NERDP on the nanosatellite sending constant unauthorized

requests. Unfortunately, there is little that can be done by network protocols to deter physical attacks such as signal jamming. The mitigation of malicious logic and resource exhaustion through requests can be mitigated by implementing an authentication system for the requests. This changes the problem of availability to a problem of integrity and confidentiality, which NERDP can provide easily.

G. CHAPTER SUMMARY

Research into nanosatellite communication standards led to the conclusion that the community lacks a standardized approach to mitigate the problems of information assurance. In researching mechanisms to mitigate these risks, NERDP was designed as a flexible solution, tailored specifically to mitigate the challenges faced by the community. A proof of concept implementation in a test bed proved the feasibility of creating such a system. Further research and performance evaluation allowed NERDP to be evaluated and compared to other protocols to determine its validity as a standard for nanosatellite communications. Profiling NERDP's behavior in both ideal and worst case scenarios, and profiling OTP encryption's strength and weaknesses demonstrates that such a protocol has a place as a nanosatellite communication protocol. The data overhead costs are reduced as the packets get larger and such a system provides reliability functionality at even lower cost than the most basic of IP protocols, UDP.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND RECOMMENDATIONS

A. INTRODUCTION

Before engaging in a full implementation of a lightweight protocol for communicating with nanosatellites over low-bandwidth communications, more development and research is needed. This thesis seeks a solution for implementing information assurance in nanosatellite communication schemes, and upon finding a lack of a standardized approach, proposes a protocol that provides all of the functionality desired from a protocol, at a low data overhead cost. Some initial implementations utilizing COTS components produce a proof-of-concept test harness that allows for initial evaluation into the feasibility of the protocol design. The experiment and design is now synthesized into a discussion on the lessons learned and the future development of the protocol.

B. MAIN CONCLUSIONS AND RECOMMENDATIONS

This thesis sought to answer three initial research questions all within the scope of nanosatellite communications operations. The questions were designed as a method for evaluating the status of the current communication schemes, and to establish overall goals for design and evaluation of the proposed protocol.

1. What are the processing, data overhead, and encryption costs of current nanosatellite communication protocols? These are an aggregate of computational time, bytes of unnecessary data in headers, and complexity of encryption mechanism.

2. Is a one-time pad approach for encryption in nanosatellite communications viable, and how does this approach compare to CSP and XTEA in terms of processing and storage costs?

3. Is there a protocol scheme to reduce the amount of data overhead and result in faster transfer times and/or a reduced number of packet exchanges than TCP?

For the first question it was hypothesized, based on the properties of OTPs, that OTP encryption mechanisms would be more efficient systems in terms of processing, data overhead, and encryption costs. In order to evaluate these criteria, an OTP encryption mechanism was developed and tested, and calculations were undertaken to estimate the different costs of OTP, AES-128, and XTEA. The development of this platform allowed us to definitively state that OTP was the most efficient mechanism in all three categories. OTP reduces the processing overhead by reducing the number of iterations and functions per bytes encrypted and does not increase the size of the encrypted data file in comparison to the unencrypted data, thus resulting in lower costs to processing power. When combined, these factors reduce the impact and cost to encrypt data in comparison to AES-128 and XTEA.

In reference to the second research question, when integrated into the NERDP protocol, by not utilizing block sizes and maintaining simple instructions, OTP encryption provides lower processing and data transfer cost than CSP and XTEA. This conclusion is drawn by studying at the number of iterations XTEA must undergo before the data is deemed secure. XTEA requires 64 iterations of 8 bytes to be considered secure, while OTP encryption only requires one iteration per byte to establish perfect secrecy [16], [20]. This was calculated and verified with the encryption platform developed. By reducing the need for padding to certain block sizes, it was also determined that XTEA could trigger the need for another packet full of padding that would otherwise be unnecessary under OTP encryption. In this regard, OTP was a much more efficient encryption system. The weakness of OTP relative to both XTEA and AES- 128 is independent of the protocol utilized to transfer the data. OTP requires truly random data as long as the total lifetime data that will be encrypted by the satellite. Depending on the usage of the satellite and its design, this could amount to large amounts of data having to be stored on a satellite constrained for memory, weight, power, and volume. Additionally, the impact of radiation affecting the stored data needs to be assessed. This is a drawback for an encryption system, since the symmetric key will not be able to be verified or altered if corrupted. Further research is needed on ways to mitigate this issue.

The final research question sought to find a protocol that provided TCP functionality at lower data overhead, transfer times, and packet exchange cost. This investigation hypothesized that NERDP would be such a protocol. Based on observed behavior and the calculations, we confirmed that NERDP results in a drastic reduction of data overhead over TCP, a result that becomes more apparent as the object sizes get larger. The results show that after an object surpasses approximately one kilobyte of data size, NERDP outperforms UDP in the total data overhead. While packet numbers for NERDP are still higher than UDP, the advantage is seen in the fact that the NERDP packets can transfer more data due to their increased data packet size. Additionally, this protocol also provides reliability that UDP lacks. Transfer time comparisons were unclear and not undertaken due to the fact that the NERDP prototype was developed in an interpreted language, Python, while TCP management and data transfer is done by the kernel. Adding these layers of abstraction skews the results and makes the NERDP prototype slower than it would be if it were developed in a lower level language. A brief analysis concludes that TCP would be theoretically faster than NERDP due to not needing to interrupt data transmission to move from frame to frame. NERDP mitigates this difference in speed with an increased payload size due to the reduced packet header. Further analysis is required to quantify the degree to which this is reduced, but initial data shows that NERDP is a feasible alternative to TCP.

C. MAIN CONTRIBUTIONS

The work presented in this thesis contributes to the current information assurance practices of nanosatellites and small satellites by profiling two common encryption methods and evaluating and analyzing the status of data confidentiality in nanosatellite and small satellite communication schemes. Research into the current state of these schemes resulted in a proposed encryption mechanism that would optimize the use of the limited processing resources available to the spacecraft.

The inquiry into information assurance culminated in research into a more efficient network transfer protocol that could better utilize the bandwidth in nanosatellites and small satellites. The research conducted into this protocol contributes to the

development and functionality of small satellites and nanosatellites by providing the communities a COTS open source design for a network protocol that is designed around the limitations of the spacecraft. Based on the designs, a lightweight proof of concept platform was created on a compressed schedule to evaluate the feasibility and performance of such a device should it be implemented. While further analysis and functionality implementation is still needed, the initial results support the conclusion that the design is sound and could provide a much needed standard for communication.

Overall the study identifies the key limitations of nanosatellites and small satellites, processing, power, and bandwidth, and draws conclusions from the pitfalls of previous communications schemes. The result is an entirely new design tailored to the specific needs of the nanosatellite and small satellite communities. The contributions are helpful to those looking for lightweight solutions where reliability, integrity, and confidentiality do not have to be sacrificed due to the bandwidth limitations.

D. FUTURE WORK

The design of the encryption mechanism and the NERDP implementation provide basic functionality that was laid out in their design specification. The current test platforms serve to provide researchers and designers with initial data and a basic working data transfer protocol, but still require more development in a lower level language to be developed into a robust COTS solution. Currently the platforms work on COTS components and can be used by anyone as the source code is made open and available. Future iterations of development need to integrate many design characteristics outlined in the design specifications in Chapter III and Chapter IV. These design specifications are not exhaustive nor are they rigid, NERDP was designed to be flexible so that designers can apply and integrate it as they see fit. Future work will develop multiple versions each with different functionality built into it so that nanosatellite and small satellite researchers will only have to pull the code from repositories and with minimal modification integrate it into their spacecraft. Before any of this development is undertaken, it is necessary to conduct thorough tests and performance evaluations to ensure the protocol is in fact

behaving as expected and supporting the hypothesis and claims made in this investigation.

E. SUMMARY

In approximately 5 months of research and developmental effort, this thesis has effectively categorized the limitations of nanosatellites and small satellite communications, characterized the current solutions developed to mitigate those limitations, and has proposed a design for a new protocol that will be provided as COTS for all members of the nanosatellite and small satellite communities. This effort has also demonstrated that such a design is sound, efficient, and can be used to improve communication performance and information assurance standards currently in use. By approaching the problem as a software problem, research and development of the Nanosatellite Encrypted Reliable Datagram Protocol resulted in a solution that will further elevate and increase the functionality of nanosatellites and small satellites.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. ONE-TIME-PAD ENCRYPTION MECHANISM TESTBED

```
# One time pad encryption mechanism test bed
#https://github.com/cabanuel/SatsThesis
# developed in Python 3.5.2
import struct
import binascii
import numpy

#                                                                        #
************************************************************************
*********************************************
# # Packing to bytes because len of int = 4 bytes, len of char = 1 byte
#                                                                        #
************************************************************************
*********************************************

# # Length of message
# messageLen = len(message)

# packedEncryptedMsg = struct.pack("{0:d}B".format(messageLen), *encryptedMsg)



def encryptMessage(message, pad):
	#
************************************************************************
*********************************************
	# Encrypt the message converting the values into int and XORing them
	#
************************************************************************
*********************************************
	encryptedMsg =[]
	for m,p in zip(message,pad):
		cipher = (ord(m)^ord(p))
		encryptedMsg.append(cipher)
	# print('inside encrypt func: ',encryptedMsg)
	return encryptedMsg

def packMessage(encryptedMsg):
	#
************************************************************************
*********************************************
```

```
        # Gotta pack the message, because python likes ints to be 4 bytes, while C and
everyone else likes 1 byte per char
        #
**********************************************************************
**********************************************
        messageLen = len(encryptedMsg)
        packedEncryptedMsg          =          struct.pack("{0:d}B".format(messageLen),
*encryptedMsg)
        return packedEncryptedMsg

def writePackedMsg(packedEncryptedMsg):
        #
**********************************************************************
**********************************************
        # Write the packed message out to a file (will be later used to send over network)
        #
**********************************************************************
**********************************************
        f = open('cipher.txt','wb')
        # encryptedMessage = bytes(encryptedMessage)
        f.write(packedEncryptedMsg)
        f.close()
        return

def readPackedMsg():
        #
**********************************************************************
**********************************************
        # open file, read message
        #
**********************************************************************
**********************************************
        f = open('cipher.txt','rb')
        encryptedMsgRead = f.read()
        f.close()
        return encryptedMsgRead

def unpackMessage(packedEncryptedMsg):
        #
**********************************************************************
**********************************************
        # unpack to use with python again
        #
**********************************************************************
**********************************************
```

```
        messageLen = len(packedEncryptedMsg)
        unpackedencryptedMessage = struct.unpack("{0:d}B".format(messageLen),
packedEncryptedMsg)
        return unpackedencryptedMessage

def unpack_ParseMessage(encryptedMsgRead):
        #
**********************************************************************
**********************************************
        # gotta make it into a list to be able to play with it
        #
**********************************************************************
**********************************************
        unpackedencryptedMessage = unpackMessage(encryptedMsgRead)
        unpackedencryptedMessageList = list(unpackedencryptedMessage)
        return unpackedencryptedMessageList

def decryptMessage(unpackedencryptedMessageList,pad):
        #
**********************************************************************
**********************************************
        # same as encrypt
        #
**********************************************************************
**********************************************
        decryptedMessage = []
        for e,p in zip(unpackedencryptedMessageList,pad):
                clearText = chr(e ^ ord(p))
                decryptedMessage.append(clearText)
        return decryptedMessage

def bitFlipper(encryptedMsgRead):
        #
**********************************************************************
**********************************************
        # Take random bits and flip them with a discrete probability. Used to measure
error propagation in simulation
        #
**********************************************************************
**********************************************
        # convert the packed, encrypted message into bits and make it a list
        binaryMsg =  bin(int.from_bytes(encryptedMsgRead, 'big'))
        binaryMsgList = list(binaryMsg)
        # print('Binary MSG LIst: ', binaryMsgList)
        print('Binary MSG before flip: ', binaryMsg)
```

```
        # Based on a probability, we can establish the probability of each bit getting
flipped

        # 0 for no change, 1 flips the bit
        elements = [0,1]
        # 1/16 bits needs to be flipped. so probability of flipping is 0.0625
        probabilities = [0.9375, 0.0625]

        # need to keep count of bits flipped for later analysis
        bitsFlipped = 0

        # need to skip the 0, and 1st bit since they  are there for python reasons
        for i in range(2, len(binaryMsgList)):
                # get a random probability (labeled coin for coin toss though probabilities
can be changed)
                coinList                                                        =
(numpy.random.choice(elements,1,p=list(probabilities))).tolist()
                coin = coinList[0]
                # if the coin is 0 then go back to the top, and increase i
                if coin == 0:
                        continue
                # else the coin is not zero, so we have to change 0 -> 1, and 1 ->0 in
position i of binaryMsgList
                if      binaryMsgList[i] == '0':
                        binaryMsgList[i] = '1'
                        bitsFlipped += 1
                        continue
                if  binaryMsgList[i] == '1':
                        binaryMsgList[i] = '0'
                        bitsFlipped += 1

        # after the bits have been flipped time to rejoin the list into a string
        binaryMsg = ''.join(binaryMsgList)
        print('binary MSG after flip: ', binaryMsg)

        # convert to ints for python because python loves ints
        binaryMsgInt = int(binaryMsg,2)

        # we then convert back to the packed char bytes that we had originally
        encryptedMsgReadFlipped = binascii.unhexlify('%x' % binaryMsgInt)

        # quick test
        # ************************************************
        print('No flip: ',encryptedMsgRead)
```

```
        print('Flip: ', encryptedMsgReadFlipped)
        if encryptedMsgReadFlipped == encryptedMsgRead:
                print('Same')
        else:
                print('Different')
        # ************************************************

        # return the encrypted packed message with some bits flipped and the counter.
        return encryptedMsgReadFlipped, bitsFlipped

def flipSubroutine(i,binaryMsgList):
        if      binaryMsgList[i] == '0':
                binaryMsgList[i] = '1'
        else:
                binaryMsgList[i] = '0'
        return binaryMsgList[i]

def tripletBitFlipper(encryptedMsgRead):
        #
********************************************************************************
*******************************************
        # Take random bits and flip them with a discrete probability. Used to measure
error propagation in simulation
        #
********************************************************************************
*******************************************
        # convert the packed, encrypted message into bits and make it a list
        binaryMsg =  bin(int.from_bytes(encryptedMsgRead, 'big'))
        binaryMsgList = list(binaryMsg)
        # print('Binary MSG LIst: ', binaryMsgList)
        print('Binary MSG before flip: ', binaryMsg)

        # Based on a probability, we can establish the probability of a bit getting flipped

        # 0 for no change, 1 flips the bit
        elements = [0,1]
        # 1/16 bits needs to be flipped. so probability of flipping is 0.0625
        probabilities = [0.9375, 0.0625]

        # need to keep count of bits flipped for later analysis
        bitsFlipped = 0

        # initialize index of the list to 2
        # need to skip the 0, and 1st bit since they  are there for python reasons ('0' and 'b')
        # thats why we start at 2
```

```
        i = 2


        while (i < len(binaryMsgList)):
                # get a random probability (labeled coin for coin toss though probabilities can be changed)
                coinList = (numpy.random.choice(elements,1,p=list(probabilities))).tolist()
                coin = coinList[0]
                # if the coin is 0 then go back to the top, and increase i
                if coin == 0:
                        i+=1
                        continue
                # else the coin is not zero, so we have to change 0 -> 1, and 1 ->0 in position i of binaryMsgList
                else:
                        # flip the first bit at position i
                        binaryMsgList[i] = flipSubroutine(i,binaryMsgList)
                        # increase thindex to i+1
                        i+=1
                        bitsFlipped+=1
                        # if i+1 < EOF then we flip it and increase i to i+2
                        if (i< len(binaryMsgList)):
                                binaryMsgList[i] = flipSubroutine(i,binaryMsgList)
                                i+=1
                                bitsFlipped+=1
                        # if i+1 suceeded in flipping, then we test if i+2 < EOF, if not then we are done and go back to while loop which will exit
                        # if i+1 failed then this will also fail and we will go back to the top of the while loop
                        if (i< len(binaryMsgList)):
                                binaryMsgList[i] = flipSubroutine(i,binaryMsgList)
                                i+=1
                                bitsFlipped+=1



                # if    binaryMsgList[i] == '0':
                #       binaryMsgList[i] = '1'
                #       bitsFlipped += 1
                #       continue
                # if  binaryMsgList[i] == '1':
                #       binaryMsgList[i] = '0'
                #       bitsFlipped += 1
```

```
        # after the bits have been flipped time to rejoin the list into a string
        binaryMsg = ''.join(binaryMsgList)
        print('binary MSG after flip: ', binaryMsg)

        # convert to ints for python because python loves ints
        binaryMsgInt = int(binaryMsg,2)

        # we then convert back to the packed char bytes that we had originally
        encryptedMsgReadFlipped = binascii.unhexlify('%x' % binaryMsgInt)

        # quick test
        # ************************************************
        print('No flip: ',encryptedMsgRead)
        print('Flip: ', encryptedMsgReadFlipped)
        if encryptedMsgReadFlipped == encryptedMsgRead:
                print('Same')
        else:
                print('Different')
        # ************************************************

        # return the encrypted packed message with some bits flipped and the counter.
        return encryptedMsgReadFlipped, bitsFlipped



def main():
        #
**********************************************************************
**********************************************
        # Set up messages needed to encrypt and the OTP
        # Make them lists of one byte of 'str'
        #
**********************************************************************
**********************************************

        # Set up the data string we want to encrypt
        # Set up the raw pad
        string                                                          =
'\x00abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890!@#$%^&*()\x00.,'
        padlong                                                         =
11111111111111111111111111111111111111111111111111111111111111111111111111111
```

```
        # TODO: Instead of hardcoding these, work on reading them from a file.


        # make each byte on the string it's own item in a list of type 'str'
        message = list(string)
        # make each byte on the string it's own item in a list of type 'str'
        pad = list(str(padlong))

        #
**************************************************************************
*****************************************
        # set up some of the arrays used
        #
**************************************************************************
*****************************************

        encryptedMsg = []
        decryptedMsg = []
        encryptedMsgRcvd = []

        # encrypt message
        encryptedMsg = encryptMessage(message,pad)

        # pack the message
        packedEncryptedMsg = packMessage(encryptedMsg)

        # write packed message to file
        writePackedMsg(packedEncryptedMsg)

        # read packed message from file
        encryptedMsgRead = readPackedMsg()
        print('Before errors introduced: ',encryptedMsgRead)
        encryptedMsgReadFlipped, bitsFlipped = tripletBitFlipper(encryptedMsgRead)

        # unpack message read
        encryptedMsgRcvd = unpack_ParseMessage(encryptedMsgRead)
        encryptedMsgRcvdFlipped = unpack_ParseMessage(encryptedMsgReadFlipped)
        # decrypt the message
        decryptedMsg = decryptMessage(encryptedMsgRcvd, pad)
        decryptedMsgFlipped = decryptMessage(encryptedMsgRcvdFlipped, pad)

        print('*'*40)

        # CHECK TO SEE IF FUNCTION CHANGED STUFF
        print('encryptedMsg: ')
```

```
        print(encryptedMsg)
        print('*'*40)
        print('packedEncryptedMsg :')
        print(packedEncryptedMsg)
        print('*'*40)
        print('encryptedMessageRead :')
        print(encryptedMsgRead)
        print('*'*40)
        print('encryptedMsgRcvd :')
        print(encryptedMsgRcvd)
        print('*'*40)
        print('decryptedMessage :')
        print(decryptedMsg)
        print('*'*40)
        print('decryptedMessageFlipped : ')
        print(decryptedMsgFlipped)
        print('*'*40)
        print('bits flipped :')
        print(bitsFlipped)


if __name__ == '__main__':
        main()
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. GROUND STATION RECEIVER TESTBED

```
#ground station implementaion testbed
#https://github.com/cabanuel/SatsThesis
#developed in python 3.5
import socket
import os
import sys
from  struct import *
from math import *

# set IP address of source machine for sending, and dummy port (just filler)

CUDP_IP = "0.0.0.0"
CUDP_PORT = 0

IP_address_src = '192.168.1.3'
IP_address_dst = '192.168.1.2'
# set the length of max read (e.g. cadet can only send 77 bytes at a time)
readLen = 77
# set the length of the actual data packet (readLen - IPv4 header - NERDP header)
dataPacketLen = readLen - 20 -4

# create a raw socket that will bind to the network interface, this will receive all raw packets at the OSI layer 3
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_RAW)

try:
   s.bind((CUDP_IP,CUDP_PORT))
except:
   print ('ERROR BINDING CHECK PRIVS')
   sys.exit()


def sendPacket(IP_address_dst,IP_address_src, packetID, packetType, payload, reqPort):

# PAYLOAD MUST BE OF TYPE BYTES SO IT CAN BE ENCODED LATER
# IP_address_* MUST BE OF TYPE STR
# packetID MUST BE OF TYPE INT
# packetType MUST BE OF TYPE STR

   packet = '';

    #THIS IS FIXED
```

```
    source_ip = IP_address_src

    # PASSED AS A PARAMETER AS STRING
    dest_ip = IP_address_dst

    # THE IP HEADER, THE IP HEADER NEVER CHANGES

    # ip header fields
    ip_ihl = 5
    ip_ver = 4
    ip_tos = 0
    ip_tot_len = 38  # kernel will fill the correct total length
    ip_id = 54321   #Id of this packet
    ip_frag_off = 0
    ip_ttl = 255
    ip_proto = socket.IPPROTO_RAW
    ip_check = 0    # kernel will fill the correct checksum
    ip_saddr = socket.inet_aton ( source_ip )   #Spoof the source ip address if you want to
    ip_daddr = socket.inet_aton ( dest_ip )

    ip_ihl_ver = (ip_ver << 4) + ip_ihl

    # the ! in the pack format string means network order
    ip_header = pack('!BBHHHBBH4s4s' , ip_ihl_ver, ip_tos, ip_tot_len, ip_id,
ip_frag_off, ip_ttl, ip_proto, ip_check, ip_saddr, ip_daddr)

#
**************************************************************************
****************************************
# This is where we determine out NERDP header and append the payload and send the
packets
# portbyte is a byte containing the source port onthe first 4 bits of the byte, and the
destination port in the last 4
# This allows for 16 (0-15) ports for source and destinations
# Each packet has the ip_header structure + the NERDP_header structure + and the
payload
# the header is 4 bytes, each other message extends that header by 3 bytes, but data
transmission strictly 4 bytes per
# header. These packets get rerouted because they don't go to port 0
#
# Currently the only reserved port is port 0, that is for ACK, SYN, REQ, MIS
# Future implementations may reserve port 1 for State of health and telemetry data
```

```
#
**********************************************************************
**************************************

   # Packet type used to request object
   if packetType == 'REQ':
      # Request must include a data port (not 0,1,2) on which the data will be sent
      reqPortByte = (str(reqPort).encode('ascii'))
      reqPortByte = pack('B',reqPort)
      data = packetType.encode('ascii') + reqPortByte + payload # REQ (3 bytes), payload
= REQUESTED PORT+objectname
      # REQ packets get sent from port 0 to port0
      srcport = 0
      dstport = 0
      # Store the source port on the upper 4 bits of the portbyte, and the destination port
on the lower 4 bits
      portByte = (srcport<<4)+dstport

      checksum = 0 # TODO: write a function to calculate checksum of payload
      # Pack the header to 4 bytes total
      NERDP_header = pack('!BHB', portByte, checksum, packetID)
      s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

   # Packet type used to acknowlege request packet
   if packetType == 'ACK':
      data = packetType.encode('ascii') + payload #payload = ACK, OTP_OFFSET (8
bytes) , OBJ_SIZE (8 bytes)
      # ACK packets get sent from port 0 to port 0
      srcport = 0
      dstport = 0
      # Store the source port on the upper 4 bits of the portbyte, and the destination port
on the lower 4 bits
      portByte = (srcport<<4)+dstport

      checksum = 0 # TODO: write a function to calculate checksum of payload
      NERDP_header = pack('!BHB', portByte, checksum, packetID)
      s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

   # Packet type used to signal end of transmission, whether from EOF or if the ground
station terminates
   if packetType == 'FIN':
      data = packetType.encode('ascii') + payload #empty payload
      # FIN packets get sent from port 0 to port 0
      srcport = 0
      dstport = 0
```

```
        # Store the source port on the upper 4 bits of the portbyte, and the destination port on the lower 4 bits
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        # Pack the header to 4 bytes total
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

    # Packet type used to synchronize and request any retransmissions of the 255 packet frame
    if packetType == 'SYN':
        data = packetType.encode('ascii') + payload #payload = SYN, NULL
        # SYN packets get sent from port 0 to port 0
        srcport = 0
        dstport = 0
        # Store the source port on the upper 4 bits of the portbyte, and the destination port on the lower 4 bits
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

    # Packet type used to indicate the missing packets and request retransmission
    if packetType == 'MIS':
        data = packetType.encode('ascii') + payload #payload = MIS, Packet numbers where each byte is one packetID
        # MIS packets get sent from port 0 to port 0
        srcport = 0
        dstport = 0
        # Store the source port on the upper 4 bits of the portbyte, and the destination port on the lower 4 bits
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        # Pack the header to 4 bytes total
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

    # Packet type used to initiate transmission of next 255 packet frame and continue data transmission (end of retransmission)
    if packetType == 'CON':
        data = packetType.encode('ascii') + payload #payload = MIS, Packet numbers where each byte is one packetID
```

```
        # CON packets get sent from port 0 to port 0
        srcport = 0
        dstport = 0
        # Store the source port on the upper 4 bits of the portbyte, and the destination port on the lower 4 bits
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        # Pack the header to 4 bytes total
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

    # Packet type used to indicate payload data packet
    if packetType == 'DAT':
        data = payload # payload is the data being sent
        # packets get sent from port 2 to requested port
        srcport = 2
        dstport = reqPort
        # Store the source port on the upper 4 bits of the portbyte, and the destination port on the lower 4 bits
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        # Pack the header to 4 bytes total
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

    if packetType == 'SRQ': #request
        data = payload.encode('ascii') # payload = 'SOHREQ'
        # SOH packets get sent from port 1 to port 1
        srcport = 1
        dstport = 1
        # Store the source port on the upper 4 bits of the portbyte, and the destination port on the lower 4 bits
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        # Pack the header to 4 bytes total
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

    if packetType == 'SRP': #response
        data = packetType.encode('ascii') + payload # payload = 'SOHRSP' + data of SOH
        # SOH packets get sent from port 1 to port 1
        srcport = 1
```

```
        dstport = 1
        # Store the source port on the upper 4 bits of the portbyte, and the destination port
on the lower 4 bits
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        # Pack the header to 4 bytes total
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

def main():
    while True:
        while True:
            print('*MAIN MENU*')
            print('1. SEND OBJ REQ AND CAPTURE DATA')
            # TODO:
            # print('2. SEND REQ AND CAPTURE SOH')
            # print('3. SEND UPL AND SEND OBJECT')

            # input the response, some input validation
            try:
                response = int(input('PLEASE ENTER NUMBER OF DESIRED ACTION: '))
                break
            except:
                print('Exiting')
                sys.exit()



        # Requesting an object from the Satellite
        if response == 1:

            # get name and requested object on data port between 3 and 15
            obj = str(input('ENTER NAME OF REQUESTED OBJECT: '))
            reqPort = int(input('ENTER NUMBER OF REQUESTED DATA PORT: '))

            # Build the REQuest packet
            packetID = 0
            packetType = 'REQ'
            payload = obj.encode('ascii')

            # send REQuest packet with object name, and the requested port
            sendPacket(IP_address_dst, IP_address_src, packetID, packetType, payload,
reqPort)
```

```
# start recieving the object
print('SWITCHING TO RECEIVING MODE')
# recieve each packet (77 bytes)
dataSent = 0
paketID = 0
totalPacketsRcvd = 0
targetPacketsRcvd = 0
recvdMsgBuffer = {}
repeatPackets = ''
#  TESTING RETRANSMISSION - remove line 241 before operation
x = 0
f = open('test.txt', 'wb')


# start data collection

# TODO: add functionality for state of health packets sent to port 0
# print('THIS IS USED FOR NON DATA TRANSFERR PACKETS')

# else it must all be data being sent to reqport
while True:

    # PAYLOAD LENGTH HERE IS SET AT 77 FOR THE MODEL USED
    # CAN BE CHANGED TO N BYTES.
    # recieve the readLen number of bytes
    packetRcvd = s.recvfrom(readLen)
    # get rid of the IP header
    packetRcvd = packetRcvd[0]
    # unpack the portbyte and get the src and dst ports
    portByte   = format(int(packetRcvd[20]),'02x')
    srcport = int(portByte[0],16)
    dstport = int(portByte[1],16)
    # get the checksum (2 bytes)
    checksum   = packetRcvd[21:23]
    # packetID # (between 0-255)
    packetID   = packetRcvd[23]
    # the rest is payload
    payload    = packetRcvd[24:]
    # get payload type
    packetType = payload[0:3].decode('ascii')


    if dstport == 0:
        if packetType == 'ACK':
```

```
                # TODO set a timer for ACK, if the next packet received

                # parse ACK
                # ACK is added to the written file
                # 0-3 ACK, 3-10 OTP offset, 10-17 obj size

                ackPayload = payload[3:19]
                # unpack the payload
                ackPayload = unpack('!QQ',ackPayload)
                # get one time pad offset and object size from payload
                OTP_OFFSET = ackPayload[0]
                OBJ_SIZE = ackPayload[1] # in bytes

                # how many packets is ground expecting
                targetPacketsRcvd = ceil(OBJ_SIZE/dataPacketLen) + 1 #total
data/packetsize + the ACK packet
                # the length of the last packet
                lastPacketLen = OBJ_SIZE%dataPacketLen
                # how many packets are in the last frame
                packetsInLastFrame = targetPacketsRcvd%256
                # get the payload and say set the ack flag so we know we have received
the ack
                # we need the ack flag to trigger ack retransmission, this is important to
get the
                # Object size and determine retransmission
                recvdMsgBuffer[packetID] = payload
                ackFlag = 1
                # we have received 1 packet
                totalPacketsRcvd += 1

            # if we receive a SYNchronization packet, means we have received 256
packets of data
            # and the satellite has NOT sent all of the object yet
            if packetType == 'SYN':
                # trigger check for missing/corrupted packets, then CON
# DELETE PACKET TEST START REMOVE BEFORE FLIGHT
                if x == 0:
                    print('***************REPEAT TEST', recvdMsgBuffer[0])
                    del recvdMsgBuffer[0]
                    totalPacketsRcvd -=1
                    x+=1
                    ackFlag = 0
# DELETE PACKET TEST END

                # if we received SYN we need to check if we are missing any packets
```

```
                # ground must have received packets 0-255
                repeatPackets = ''
                for i in range(256):
                    # use the key of the received message buffer dictionary as the packet ID
                    if i in recvdMsgBuffer:
                        # if packet exists, append '0' to a string
                        repeatPackets += '0'
                        continue
                    else:
                        # if packet doesnt exist, append '1'
                        repeatPackets += '1'
                # we now have a string of 0's and 1's of len 256 where the position on the
list
                # determines the packetID
                if '1' in repeatPackets:
                    # if there's a '1' in this we trigger retransmission
                    i = 0
                    missingPack = []
                    while i < 32:
                        # we take 8 characters at a time, treat them as int and pack them
                        # into a byte. this way we can get 256 packets packed into 32 bytes
                        missingPack.append(int(repeatPackets[i*8:i*8+8],2))
                        i+=1
                    # pack the first byte (8 packets per byte)
                    payload = pack('!B', missingPack[0])
                    # pack the rest of the 32 bytes
                    for i in range(1,32):
                        payload += pack('!B', missingPack[i])

                    # send the MIS packet requesting retransmission with the 32 bytes of
                    # missing packet information. Packet ID = 255, ports are 0 and 0
                    packetID = 255
                    sendPacket(IP_address_dst, IP_address_src, packetID, 'MIS', payload,
0)
                    continue
                else:
                    # write the packets to file, send a CON packet, and get the next 255
packets
                    for i in recvdMsgBuffer:
                        # write the packets (in order including the ack)
                        f.write(recvdMsgBuffer[i])
                    # clear the dictionary
                    recvdMsgBuffer= {}
                    # send a CONtinute packet requesting the next frame of packets
                    packetID = 255
```

```
                payload = '0' #NULL payload
                payload = payload.encode('ascii')
                sendPacket(IP_address_dst, IP_address_src, packetID, 'CON', payload,
0)
                continue


                # If ACK flag is missing, we just treat it as data packet for
retransmission
                # IFF we received at least 256 data packets, we determine this by
receiving the SYN packet


            # If we receive a FINish packet, it means the satellite has sent less than 256
packets in this frame
            # and has reached EOF
            if packetType == 'FIN':
               repeatPackets = ''

# DELETE PACKET TEST START, REMOVE BEFORE FLIGHT
               if x == 0:
                  print('**********REPEAT TEST', recvdMsgBuffer[0])
                  del recvdMsgBuffer[0]
                  totalPacketsRcvd -=1
                  x+=1
                  ackFlag = 0
# DELETE PACKET TEST END
               # if we are missing the ACK packet, and the flag has not been triggered
indicating
               # we are still on the first frame, we must request it alone
               # we cannot do retranmission on frames less than 256 packets without the
OBJ_SIZE
               if (('0' not in recvdMsgBuffer) and (totalPacketsRcvd < 256) and (ackFlag
== 0)):
                  # repeating ACK
                  # only '1' in retranmission 256 string is ACK flag
                  repeatPackets +='1'
                  repeatPackets += '0'*255

                  i = 0
                  missingPack = []
                  while i < 32:
                     # we take 8 characters at a time, treat them as int and pack them
                     # into a byte. this way we can get 256 packets packed into 32 bytes
                     missingPack.append(int(repeatPackets[i*8:i*8+8],2))
```

```
                    i+=1
                # pack the first one
                payload = pack('!B', missingPack[0])
                # pack the rest
                for i in range(1,32):
                    payload += pack('!B', missingPack[i])
                # send MIS packet
                packetID = 255
                sendPacket(IP_address_dst, IP_address_src, packetID, 'MIS', payload,
0)
                continue


            # if we get a FIN and we hace the ACK packet we can then just trigger
retransmission as normal
            for i in range(packetsInLastFrame):
                # use the key of the received message buffer as the packet ID
                if i in recvdMsgBuffer:
                    # if packetID is there, then we append '0' to a string
                    repeatPackets += '0'
                    continue
                else:
                    # if not we append '1'
                    repeatPackets += '1'
            # since we didnt receive a full frame, we must then pad the remainder
packets
            # with 0's to reach the 32 Bytes
            while len(repeatPackets) < 256:
                repeatPackets += '0'

            # is retranmsission needed?
            if '1' in repeatPackets:
                # if there is a 1, yes
                i = 0
                missingPack = []
                while i < 32:
                    # we take 8 characters at a time, treat them as int and pack them
                    # into a byte. this way we can get 256 packets packed into 32 bytes
                    missingPack.append(int(repeatPackets[i*8:i*8+8],2))
                    i+=1

                # pack the first byte
                payload = pack('!B', missingPack[0])

                # pack the rest of the 32 bytes
```

```
                    for i in range(1,32):
                        payload += pack('!B', missingPack[i])

                    # Send MIS packet requesting retransmission
                    packetID = 255
                    sendPacket(IP_address_dst, IP_address_src, packetID, 'MIS', payload,
0)
                    continue
                else:
                    # write packets to file,  send a FIN packet and exit
                    for i in recvdMsgBuffer:
                        f.write(recvdMsgBuffer[i])
                    packetID = 255
                    payload = '0' #NULL payload
                    payload = payload.encode('ascii')
                    sendPacket(IP_address_dst, IP_address_src, packetID, 'FIN', payload,
0)
                    break


            # if dstport != 0 then it is the reqport (for now) and it is data and we must
append it to the dict
            print('SAVING')
            recvdMsgBuffer[packetID] = payload
            totalPacketsRcvd +=1

        # close file
        f.close()
        print('DONE')
        # reset ack flag
        ackFlag = 0




# ********************************
# ********************************
# Begin Ground logic
# ********************************
# ********************************
if __name__ == '__main__':
    main()
```

APPENDIX C. SATELLITE RECEIVER TESTBED

```
#satellite implementaion testbed
#https://github.com/cabanuel/SatsThesis
#developed in python 3.5
import socket
import os
import select
from  struct import *

#
********************************************************************************
*****************************************
# Establish some parameters from user:
#
********************************************************************************
*****************************************


# set IP address of source machine for sending, and dummy port (just filler to test on
VMs on same network)

CUDP_IP = "0.0.0.0"
CUDP_PORT = 0

IP_address_src = '192.168.1.2'
IP_address_dst = '192.168.1.3'
# set the length of max read (e.g. cadet can only send 77 bytes at a time)
readLen = 77
# set the length of the actual data packet (readLen - IPv4 header - NERDP header)
dataPacketLen = readLen - 20 -4
# to vary the transmission rate we establish the timeout expecter per packet
# this is the time we expect for a packet to require for roundtrip
# THIS IS VERY DEPENDENT ON RADIO, HARDWARE, ETC. ADJUST FOR
YOUR DEVICE
packetDelay = 0.5
#
********************************************************************************
*****************************************
#
********************************************************************************
*****************************************
```

```
# create a raw socket that will bind to the network interface, this will receive all raw packets at the OSI layer 3

s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_RAW)

try:
   s.bind((CUDP_IP,CUDP_PORT))
except:
   print ('ERROR BINDING CHECK PRIVS')
   sys.exit()


def sendPacket(IP_address_dst,IP_address_src, packetID, packetType, payload, reqPort):

# PAYLOAD MUST BE OF TYPE BYTES SO IT CAN BE ENCODED LATER
# IP_address_* MUST BE OF TYPE STR
# packetID MUST BE OF TYPE INT
# packetType MUST BE OF TYPE STR

# This is the structure for sending ALL types of packets, taking in only certain parameters
# This will also fix the IP header and take charge of sending the IP packet
# For testing in VM network, 20 bytes are lost out of the 77 target len to the IP header
# This can be avoided in the radio operation, since the packet transfer will be taken care of
# by something like AX.25 protocol

   packet = '';

    #THIS IS FIXED
   source_ip = IP_address_src

   # PASSED AS A PARAMETER AS STRING
   dest_ip = IP_address_dst

   # ip header fields
   ip_ihl = 5
   ip_ver = 4
   ip_tos = 0
   ip_tot_len = 38  # kernel will fill the correct total length
   ip_id = 54321   #Id of this packet
   ip_frag_off = 0
   ip_ttl = 255
   ip_proto = socket.IPPROTO_RAW
   ip_check = 0    # kernel will fill the correct checksum
   ip_saddr = socket.inet_aton ( source_ip )   #Spoof the source ip address if you want to
```

```
    ip_daddr = socket.inet_aton ( dest_ip )

    ip_ihl_ver = (ip_ver << 4) + ip_ihl

    # the ! in the pack format string means network order
    ip_header = pack('!BBHHHBBH4s4s' , ip_ihl_ver, ip_tos, ip_tot_len, ip_id,
ip_frag_off, ip_ttl, ip_proto, ip_check, ip_saddr, ip_daddr)


    # THE IP HEADER, THE IP HEADER NEVER CHANGES

#
**************************************************************************
****************************************
# This is where we determine out NERDP header and append the payload and send the
packets
# portbyte is a byte containing the source port onthe first 4 bits of the byte, and the
destination port in the last 4
# This allows for 16 (0-15) ports for source and destinations
# Each packet has the ip_header structure + the NERDP_header structure + and the
payload
# the header is 4 bytes, each other message extends that header by 3 bytes, but data
transmission strictly 4 bytes per
# header. These packets get rerouted because they don't go to port 0
#
# Currently the only reserved port is port 0, that is for ACK, SYN, REQ, MIS
# Future implementations may reserve port 1 for State of health and telemetry data

#
**************************************************************************
****************************************

    # Packet type used to request object
    if packetType == 'REQ':
        # Request must include a data port (not 0,1,2) on which the data will be sent
        reqPortByte = (str(reqPort).encode('ascii'))
        reqPortByte = pack('B',reqPort)
        data = packetType.encode('ascii') + reqPortByte + payload # REQ (3 bytes), payload
= REQUESTED PORT+objectname
        # REQ packets get sent from port 0 to port0
        srcport = 0
        dstport = 0
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
```

```
        # Pack the header to 4 bytes total
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

    # Packet type used to acknowlege request packet
    if packetType == 'ACK':
        data = packetType.encode('ascii') + payload #payload = ACK, OTP_OFFSET (8 bytes) , OBJ_SIZE (8 bytes)
        # ACK packets get sent from port 0 to port 0
        srcport = 0
        dstport = 0
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

    # Packet type used to signal end of transmission, whether from EOF or if the ground station terminates
    if packetType == 'FIN':
        data = packetType.encode('ascii') + payload #empty payload
        # FIN packets get sent from port 0 to port 0
        srcport = 0
        dstport = 0
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        # Pack the header to 4 bytes total
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

    # Packet type used to synchronize and request any retransmissions of the 255 packet frame
    if packetType == 'SYN':
        data = packetType.encode('ascii') + payload #payload = SYN, NULL
        # SYN packets get sent from port 0 to port 0
        srcport = 0
        dstport = 0
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

    # Packet type used to indicate the missing packets and request retransmission
```

```
    if packetType == 'MIS':
        data = packetType.encode('ascii') + payload #payload = MIS, Packet numbers where
each byte is one packetID
        # MIS packets get sent from port 0 to port 0
        srcport = 0
        dstport = 0
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        # Pack the header to 4 bytes total
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

    # Packet type used to initiate transmission of next 255 packet frame and continue data
transmission (end of retransmission)
    if packetType == 'CON':
        data = packetType.encode('ascii') + payload #payload = MIS, Packet numbers where
each byte is one packetID
        # CON packets get sent from port 0 to port 0
        srcport = 0
        dstport = 0
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        # Pack the header to 4 bytes total
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

    # Packet type used to indicate payload data packet
    if packetType == 'DAT':
        data = payload # payload is the data being sent
        # packets get sent from port 2 to requested port
        srcport = 2
        dstport = reqPort
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        # Pack the header to 4 bytes total
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

    if packetType == 'SRQ': #request
        data = payload.encode('ascii') # payload = 'SOHREQ'
        # SOH packets get sent from port 1 to port 1
        srcport = 1
```

```
        dstport = 1
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        # Pack the header to 4 bytes total
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

    if packetType == 'SRP': #response
        data = packetType.encode('ascii') + payload # payload = 'SOHRSP' + data of SOH
        # SOH packets get sent from port 1 to port 1
        srcport = 1
        dstport = 1
        portByte = (srcport<<4)+dstport

        checksum = 0 # TODO: write a function to calculate checksum of payload
        # Pack the header to 4 bytes total
        NERDP_header = pack('!BHB', portByte, checksum, packetID)
        s.sendto(ip_header+NERDP_header+data, (IP_address_dst, dstport));

def main():
    while True:
        print('*LISTENING...*')
        # recieve the readLen number of bytes
        packetRcvd = s.recvfrom(readLen)
        # get rid of the IP header
        packetRcvd = packetRcvd[0]
        # unpack the portbyte and get the src and dst ports
        portByte   = format(int(packetRcvd[20]),'02x')
        srcport = int(portByte[0],16)
        dstport = int(portByte[1],16)
        # get the checksum (2 bytes)
        checksum   = packetRcvd[21:23]
        # packetID # (between 0-255)
        packetID   = packetRcvd[23]
        # the rest is payload
        payload    = packetRcvd[24:]

        # initialize the OneTimePad Offset
        # TODO: add option for encryption in ACK packet
        OTP_OFFSET = 0

        # If it is a control packet it came to port 0
        if dstport == 0:
            # All control packets in the payload have a 3 letter designation (ACK, MIS...)
```

```
packetType = payload[0:3].decode('ascii')

# If we get a request we need to process it
if packetType == 'REQ':
    # we need to send ACK
    # obtain the requested port
    reqPort = payload[3]
    objReq  = payload[4:].decode('ascii') #object requested
    objReqSizeDec = os.stat(objReq).st_size #object requested size


    # pack the objsize and the OTP offset as long unsigned ints
    payload = pack('!QQ', OTP_OFFSET, objReqSizeDec)
    # send the ACK
    sendPacket(IP_address_dst, IP_address_src, 0, 'ACK', payload, 0)

    # ***************************************************************
    # TIME TO SEND DATA
    # ***************************************************************
    # initialize counters
    dataSent = 0
    packetID = 0
    # open the rquested file as a read only
    f = open(objReq, 'rb')
    # initialize data sent dictionary
    packetSentBuff = {}
    # need to store the ACK packet with the 3 letter designator in the payload
    packetType = 'ACK'
    # used to say this is the first frame in case of needed ACK retransmission
    ackFlag = 0
    # save the ACK packet in the dictionary
    packetSentBuff[packetID] = packetType.encode('ascii') + payload
    # increase packetID #
    packetID +=1
    print('*SENDING DATA*')
    while dataSent < objReqSizeDec:

        # if we havent sent 255 packets we send data
        if packetID < 256:
            # read (in the test case 55 bytes) data from file
            payload = f.read(dataPacketLen)
            # save it to the dictionary first
            packetSentBuff[packetID] = payload
            # send the data packet to the requested port
```

```
                sendPacket(IP_address_dst, IP_address_src, packetID, 'DAT', payload,
reqPort)
                # update counters
                dataSent += dataPacketLen
                packetID += 1
            # if we have already sent the 255 packets and we have not sent the total
object size
            # we must send a SYN
            else:
                # counter for the number of SYN retransmits
                synCounter = 0
                while True:
                    # SYN packet retranmsission logic (timeout = packetDelay * number of
packets sent +1)
                    if synCounter == 0:
                        # send SYN
                        # SYN packet has a null payload
                        payload = '0'
                        payload = payload.encode('ascii')
                        # increase the synCounter
                        synCounter +=1
                        # Send the SYN packet to port 0
                        sendPacket(IP_address_dst, IP_address_src, 255, 'SYN', payload, 0)
                        # long wait for the retransmit
                        synRetransmit = select.select([s],[],[packetDelay*257])

                    if 0 < synCounter < 3:
                        # send SYN
                        # SYN packet has a null payload
                        payload = '0'
                        payload = payload.encode('ascii')
                        # increase the synCounter
                        synCounter +=1
                        # Send the SYN packet to port 0
                        sendPacket(IP_address_dst, IP_address_src, 255, 'SYN', payload, 0)
                        # for retransmits we want short wait
                        synRetransmit = select.select([s],[],[packetDelay*2])

                    if synCounter >= 3:
                        print('MAX RETRANSMITS REACHED')
                        break
                    # if we dont get data within the timeout we retransmit 2 more syns at
lower wait

                    # if we dont get any data, we go back to send the syn
```

```
                if not synRetransmit[0]:
                    continue

                # if we do get data, we  then recieve it and process the packet. Every SYN retransmit will have a short wait

                # TODO make the syn after MIS packet retransmission dynamic

                # listen for MIS packet
                packetRcvd = s.recvfrom(readLen)
                # again get rid of IP header stuff
                packetRcvd = packetRcvd[0]
                # port byte is going to be zero
                # TODO write a check to make sure that the ports are zero
                portByte   = format(int(packetRcvd[20]),'02x')
                # checksum for integrity
                checksum   = packetRcvd[21:23]
                # packet type is by default 255
                packetID   = packetRcvd[23]
                # payload of MIS is 32 bytes
                payload    = packetRcvd[24:]


                # the MIS/CONT segment of the payload
                packetType = payload[0:3].decode('ascii')
                missingPackets = payload[3:]

                # first we check that we didnt get a CONtinue message. if CON we are done retransmitting

                if packetType == 'CON':
                    # reset the packet ID
                    packetID = 0
                    # if we get some message with an ACK in it, we will treat it as data
                    ackFlag = 1
                    break

                if packetType == 'FIN':
                    # if the ground wants to terminate the transmission, this will trigger an end handshake
                    dataSent = objReqSize
                    break

                # every 8 bits inidcates 1 packet
                # if bit n is 1 it means packet n was missing and needs
```

```
                    # to be retransmitted, if 0 no retransmission.
                    # need to build a list of 1 and 0s of packets that need to be
                    # retransmitted. Index n will be the packet number
                    # 256 bits (n = 0 through 255, where n =0 is the ack packet on first
                    # session of 256 packets)

                    # set up index for packets
                    i = 0
                    missingPacketsBin = ''
                    while i < len(missingPackets):
                        # take byte number i, convert it to binary of type str in format
                        # format takes the integer converts it to binary,
                        missingPacketsBin = missingPacketsBin +
format(int(missingPackets[i]), '08b')
                        # now we increase the counter
                        i += 1

                    i = 0
                    while i < len(missingPacketsBin):
                        if missingPacketsBin[i] == '1':
                            # send the missing packets
                            payload = packetSentBuff[i]
                            # if the first frame and ACK is missing we must retransmit it by
itself to port 0
                            if payload[0:3].decode('ascii') == 'ACK' and ackFlag == 0:
                                packetID = 0
                                sendPacket(IP_address_dst, IP_address_src, packetID, 'DAT',
payload, 0)

                            else:  #otherwise it's just data
                                    packetID = i
                                    sendPacket(IP_address_dst, IP_address_src, packetID, 'DAT',
payload, reqPort)
                        i+=1
                                # after this go back to the SYN
                    # we either get a MIS request or a CON request


            while True:
                # send FIN
                # if we have sent the total size of the object we land here
                # FIN has null payload
                 payload = '0'
                 payload = payload.encode('ascii')
                 sendPacket(IP_address_dst, IP_address_src, 255, 'FIN', payload, 0)
```

```
            # listen for MIS packet
            packetRcvd = s.recvfrom(readLen)
            # again get rid of IP header stuff
            packetRcvd = packetRcvd[0]
            # port byte is going to be zero
            # TODO write a check to make sure that the ports are zero
            portByte   = format(int(packetRcvd[20]),'02x')
            # checksum for integrity
            checksum   = packetRcvd[21:23]
            # packet type is by default 255
            packetID   = packetRcvd[23]
            # payload of MIS is 32 bytes
            payload    = packetRcvd[24:]

            # the MIS/CONT segment of the payload
            packetType = payload[0:3].decode('ascii')
            missingPackets = payload[3:]

            # first we check that we didnt get a CONtinue message. if CON we are done
retransmitting

            if packetType == 'FIN':
               print('FIN RECEIVED')
               # reset the packet ID
               packetID = 0
               break

            # every 8 bits inidcates 1 packet
            # if bit n is 1 it means packet n was missing and needs
            # to be retransmitted, if 0 no retransmission.
            # need to build a list of 1 and 0s of packets that need to be
            # retransmitted. Index n will be the packet number
            # 256 bits (n = 0 through 255, where n =0 is the ack packet on first
            # session of 256 packets)


            # TODO: this means the packet has a MIS tag and port 0
            # useful for threading (future work)


            # set up index for packets
            i = 0
            missingPacketsBin = ''
            while i < len(missingPackets):
```

```
                # take byte number i, convert it to binary of type str in format
                # format takes the integer converts it to binary,
                missingPacketsBin = missingPacketsBin + format(int(missingPackets[i]),
'08b')
                # now we increase the counter
                i += 1
            # after getting the 32 bytes, and convering them to binary, we iterate over
            # the string treating the index as the index for packet. if i == 1, then
            # we go back to the dictionary and retransmit. if packets retransmitted ==0
            # we set packetID = 0, purge the dictionary, and send the next 255 packets
            # of data


            i = 0
            while i < len(missingPacketsBin):
                if missingPacketsBin[i] == '1':
                    # send the missing packets
                    payload = packetSentBuff[i]
                    # if the first frame and ACK is missing we must retransmit it by itself to
port 0
                    if payload[0:3].decode('ascii') == 'ACK' and ackFlag == 0:
                        packetID = 0
                        sendPacket(IP_address_dst,    IP_address_src,    packetID,    'DAT',
payload, 0)

                    else:  #otherwise send all of thedata
                            packetID = i
                            sendPacket(IP_address_dst,    IP_address_src,    packetID,    'DAT',
payload, reqPort)
                i+=1
            # after this go back to the SYN
            # we either get a MIS request or a CON request




# ********************************
# ********************************
# Begin SAT logic
# ********************************
```

```
# ********************************
if __name__ == '__main__':
    main()
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. GITHUB REPOSITORY FOR CODE

The source code and information on NERDP can be found at:

https://github.com/cabanuel/SatsThesis

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] R. Sandau, "Status and trends of small satellite missions for Earth observation," *Acta Astronautica*, vol. 66, no. 1-2, pp. 1–12, 2010.

[2] B. Klofas, J. Anderson and K. Leveque, "A survey of CubeSat communication systems," in *5th Annual CubeSat Developers' Workshop*, 2008.

[3] P. Muri and J. McNair, "A survey of communication sub-systems for intersatellite linked systems and CubeSat missions," *Journal of Communications,* vol. 7, no. 4, pp. 290–308, 2012.

[4] W. A. Beech, D. E. Nielsen and J. Taylor, "AX.25 link access protocol for amateur packet radio version 2.2," Tucson Amateur Packet Radio Corp., Richardson, TX,1997.

[5] "CubeSat Space Protocol: A small network-layer delivery protocol designed for CubeSats," Aalborg University, 2008. [Online]. Available: https://github.com/libcsp/libcsp. [Accessed 3 July 2017].

[6] O. N. Challa, G. Bhat and J. McNair, "CubeSec and GndSec: a lightweight security solution for CubeSat communications," in *26th Annual AIAA/USU Conference on Small Satellites*, 2012.

[7] Radar Systems Panel of the IEEE Aerospace & Electronic Systems Society, "IEEE standard for letter designations for radar-frequency bands," The Institute of Electrical and Electronics Engineers, New York, 2002.

[8] B. Kolfas, "CubeSat radios: from kilobits to megabits," in *Ground System Architectures Workshop,* 2014.

[9] D. Selva and D. Krejci, "A survey and assessment of the capabilities of CubeSats for Earth observation," *Acta Astronautica*, vol. 74, pp. 50–68, 2012.

[10] Information Sciences Institute, University of Southern California, "RFC 791 Internet Protocol: DARPA Internet Program Protocol Specification," Defense Advanced Research Projects Agency, Arlington, Virginia, 1981.

[11] Information Sciences Institute, University of Southern California, “RFC 793 Transmission Control Protocol: DARPA Internet Program Protocol Specification,” Defense Advanced Research Projects Agency, Arlington, Virginia, 1981.

[12] J. Postel, “RFC 768 User Datagram Protocol Internet Standard,” Defense Advanced Research Projects Agency, Arlington, Virginia, 1980.

[13] L. Eggert and F. Gont, “RFC 5482 TCP user timeout option,” Internet Engineering Task Force, Fremont, California, 2009.

[14] M. Swartout, “The first one hundred CubeSats: a statistical look,” *Journal of Small Satellites,* vol. 2, no. 2, pp. 213–233, 2013.

[15] G. Hunyadi, D. M. Kumpar, S. Jepsen, B. Larsen and M. Obland, “A commercial off the shelf (COTS) packet communications subsystem for the Montana EaRth Orbiting Pico-Explorer (MEROPE) CubeSat,” *Aerospace Conference Proceedings IEEE,* vol. 1, pp. 1–1, 2002.

[16] D. J. W. Roger M. Needham, “Tea extensions,” Cambridge University, Cambridge, UK, 1997.

[17] J. Lu, “Related-key rectangle attack on 36 rounds of the XTEA block cipher,” *International Journal of Information Security,* vol. 8, no. 1, pp. 1–11, 2009.

[18] Y. Ko, S. Hong, W. Lee, S. Lee and J.-S. Kang, “Related key differential attacks on 27 rounds of XTEA and full-round GOST,” in *Fast Software Encryption, 11th International Workshop*, 2004.

[19] K. Burda, “Error propagation in various cipher block modes,” *International Journal of Computer Science and Network Security,* vol. 6, no. 11, pp. 235–239, 2006.

[20] J. Katz and Y. Lindell, *Introduction to modern cryptography*, 2nd ed., Boca Raton, FL: Taylor and Francis Group, 2015.

[21] G. S. Vernam, “Secret signaling system.” U.S. Patent U.S. 1310719 A, 22 July 1919.

[22] D. Kahn, *Codebreakers: The Story of Secret Writing*, New York: Scribner, 1967.

[23] Intel Corporation, *Intel 64 and IA-32 architectures software developer's manual*, vol. 2, 2017. Intel Corporation, Mountain View, CA.

[24] National Institute of Standards and Technology, "Specification for the advanced encryption standard (AES)," National Institute of Standards and Technology, 2001.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California